Integrated Behavior Modeling of Space-Intensive Mechatronic Systems

Benjamin Hummel

Technische Universität München

Institut für Informatik
der Technischen Universität München

Integrated Behavior Modeling of Space-Intensive Mechatronic Systems

Benjamin Hummel

Vollständiger Abdruck der von der Fakultät für Informatik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitzender:	Univ.-Prof. Dr. Johann Schlichter
Prüfer der Dissertation:	
1.	Univ.-Prof. Dr. Dr. h.c. Manfred Broy
2.	Univ.-Prof. Dr. Wilhelm Schäfer, Universität Paderborn

Die Dissertation wurde am 25. August 2010 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 25. Januar 2011 angenommen.

Abstract

Many of today's systems perform their tasks by a complex combination of electrical and mechanical effects with programmable logic controllers. Such systems are commonly referred to as mechatronic or cyber-physical systems. For a large class of such systems, the notion of space is at the very core of their mode of operation, as they measure and affect the spatial relationship of physical objects. Examples of such systems are commonly found in the domain of factory automation and machine tools, where products are transported, manipulated, and assembled.

While there are many different models for capturing static and structural aspects of those systems in mechanical engineering, and computer science provides an abundance of behavior models for pure software systems, neither of these can describe and explain the operation of such systems when used in isolation. Full understanding of such a system can only be achieved by using a model that integrates the spatial and the behavioral view. Combined approaches exist, but are usually limited to the connection of different models by structural links without clearly defined semantics.

This lack of suitable modeling techniques has a practical impact, as the increasing complexity of systems built today pushes traditional development processes to their limits. In these processes the disciplines of mechanics, electrics, and software usually act isolated from each other, which inhibits taking advantage of the full potential of mechatronic solutions. One ingredient to overcome this separation of engineering disciplines is an integrated abstract model that serves as a common language for documentation and logic design of the system. By modeling and simulating the system on an abstract level, design alternatives can be explored and a common interdisciplinary understanding of the internal workings of the system can be fostered.

This thesis proposes an integrated model for space-intensive mechatronic systems that tightly couples both the structural spatial aspects and the temporal and logic behavior of a system. Compared to other approaches, strong emphasis is on the semantic meaning and mathematical foundation of the model. Such a modeling theory not only contributes to the academic discussion on suitable models but also provides a solid basis for formal analysis and construction of modeling and engineering tools.

To evaluate the adequateness of the model from an engineering perspective, this thesis also discusses the operationalization of the theoretical model and describes means by which the gap to the practical application can be closed. The actual tool prototype that implements the model is used in two case studies with machine tool vendors to investigate to which extent the behavior of real-world systems can be captured by the model.

Acknowledgments

The result of a task that spans several years, such as this thesis, is never the success of a single person, but rather depends on a multitude of supporters to whom I am deeply grateful. First of all, I want to thank my supervisor Manfred Broy for the opportunity to work in his group and his support over all the years. The impressive working environment at his chair and his ability to ask the right questions at the right time contributed greatly to this thesis. I also want to thank Wilhelm Schäfer for accepting to be my co-supervisor and for the insightful and enjoyable discussion we had during my visit in Paderborn.

I am grateful for my colleagues' support over the years, for all I could learn from them, and the many (sometimes heated) debates and discussions we had. I especially want to mention Jewgenij Botaschanjan, Peter Braun, Florian Deißenböck, Martin Feilkas, Alexander Harhurin, Lars Heinemann, Markus Herrmannsdörfer, Florian Hölzl, Elmar Jürgens, Birgit Penzenstadler, Silke Müller, Bernhard Schätz, Judith Thyssen, Stefan Wagner, and Sebastian Winter, who all impacted my work in one or another way. It was and still is a pleasure to work with all of you! Special thanks go to Florian Deißenböck, who endured the tedious task of reading the largest part of my thesis, and to Judith Thyssen, who read the core chapters (and is the only person I know, who not only reads all formulas but also spots the mistakes in them).

Several students contributed to the tool prototype and its underlying framework, and I want to thank Christoph Döbber, Christoph Klaaßen, Daniela Steidl, and Andreas Wandinger for their work.

Furthermore, I thank Alexander Lindworsky from the engineering department for teaching me a lot about mechanical engineering and the automation domain. We collaborated in multiple projects and his different perspective often lead to new insights. In this context, I also want to thank the industry partners of the AutoVIBN project for the many discussions we had during the milestone meetings. Especially Jürgen Franz and his critical questions helped me to better understand the subject matter and the practitioner's view.

For their ongoing support I want to thank my family and friends. They helped me to put everything into perspective when stuck with my work and endured me even when being slightly off. My father also deserves a big *thank you* for checking my spelling in the final version.

Thanks also go to the local Aikidoka at Ismaning for helping with the practical aspects of classical mechanics; there is no better way to relax after work than approaching the mat at high velocity.

Contents

1 Introduction

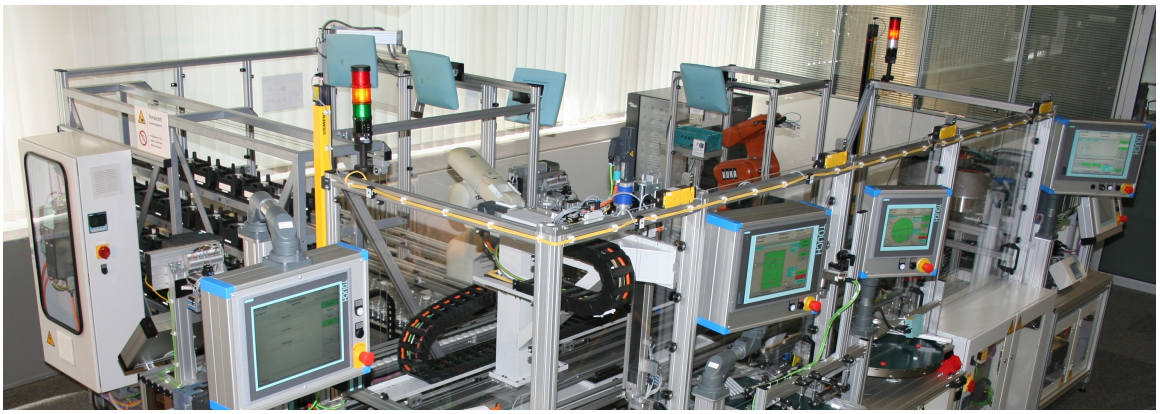
Many of today's software systems interface not only with other software systems or the user, but integrate electric and mechanic devices to form complex systems operating in the real world. Prominent examples for these integrated systems, commonly referred to as *mechatronic systems*, can be found in the automotive, avionic, and factory automation domains. This thesis will focus on the automation domain, where transportation, modification, and assembly of rigid products are the main tasks performed by mechatronic systems.

Despite the mechatronic character of modern factory automation systems the development processes for these systems are often still focused on mechanical engineering, which has a leading position. The development process is built up sequentially, with one discipline building upon the results of the previous one. This is described as "throw it over the wall" approach in [SW07]. In most companies the mechanical engineers are defining the functions of a machine at the beginning of the development process. Subsequently the machine is designed by different departments, which complement the previous models by their own artifacts, such as electric wire plans or software code. However, using this sequential process misses many opportunities for cost reduction or improvement from interdisciplinary cooperation.

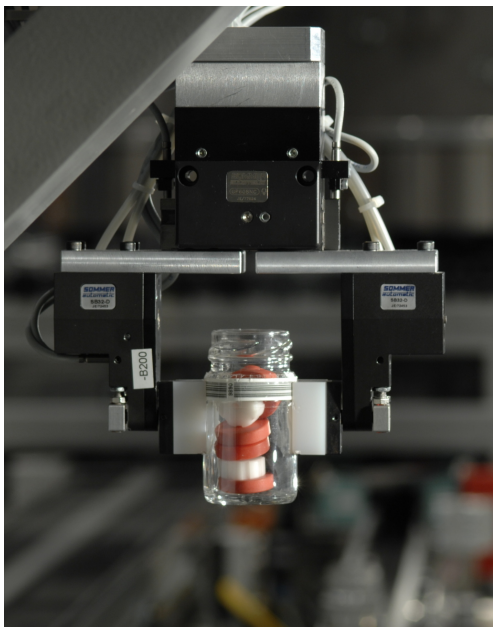
1.1 Motivating Example

An example of an automation system is the Siemens SmartAutomation plant (SmA), which was constructed as a prototype for various research activities in the automation domain. As can be seen in Figure 1.1(a) such a system typically consists of several user interfaces (often realized as touch screens), different sensors and actors (such as the robots in the background) and software controllers which supervise all of these elements. Based on production jobs initiated by an operator the machine fills small glass bottles with gears and metal disks (*c.f.*, Figure 1.1(b)). Different stations deal with subtasks such as capping and uncapping the bottles, filling in the correct parts into the bottle, palettizing the bottles, or checking whether the bottle's contents are as expected.

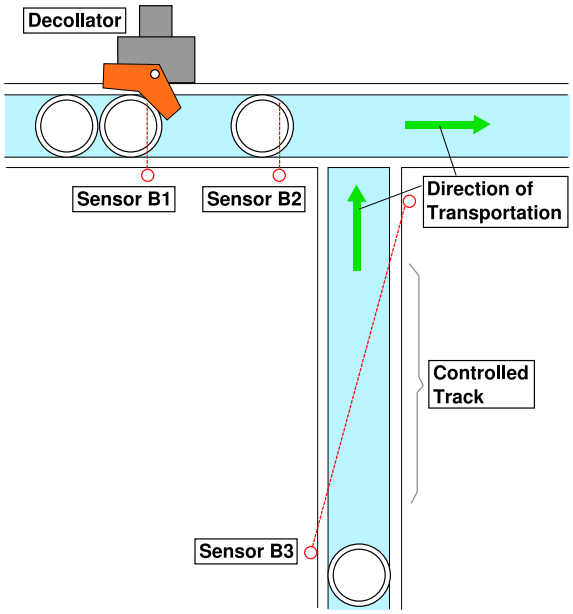
One benefit of the SmartAutomation system is that specifications and documentation are available comparatively easy. Parts of the specification [ESMS06] deal with the transportation subsystem, which routes the bottles to the stations of the machine. As the order in which a bottle visits these stations depends on the production job, the transportation system is realized with multiple belt conveyors connected by switch and junction points, thus allowing different paths for each bottle. Typical problems in the transportation system are the identification of bottles based on their bar



(a) Overview of the system



(b) Detailed view of the transported goods



(c) Schematic drawing of a junction in the transportation subsystem (adapted and translated from [ESMS06])

Figure 1.1: Siemens SmartAutomation plant (SmA)
Image source: Siemens AG, Sector Industry

```
1  U     M4.1            --  if mode is set to automatic
2  U     E412.1          --  and the input corresponding to B3 reports true
3  L     S5T#1S          --      (delay used in next command is 1 second)
4  SE    T41             --  then set T41 to true after 1 second
5                        --      (unless condition becomes false during this period)
6
7                        --  (calculation of T40 in next line is omitted here)
8  U     T40             --  if a bottle is waiting at the decollator
9  U     T41             --  and the other belt is free (T41 calculated above)
10 UN    E412.7          --  and the input corresponding to B2 is false
11 =     #Transition_OK      --  store expression's result in variable
12
13 U     M4.1            --  if mode is set to automatic
14 UN    M5.2            --  and we are not at the end of a cycle
15 U     #Transition_OK      --  and stored expression from above is true
16 S     DB331.DBX0.0        --  then set the state variable corresponding to
17                           --  the state opening the decollator
```

Figure 1.2: Software realization of a part of the congestion control. The *instruction list (IL)* code shown is based on the original code but was slightly simplified and translated into English. Also the comments are more verbose than in the original code. To understand the code it is important to know that the entire instruction list is executed cyclically as fast and often as possible.

code label, or avoidance of congestion. On Page 44, the specification describes congestion control for one of the junctions in more detail using the the schematic drawing replicated in Figure 1.1(c) and the following text[1]:

> *Congestion control on the transport sections is used to avoid jamming of bottles at track junctions. Therefore the main track is controlled in front of the junction over a long distance using a light barrier (*Sensor B3*). If this track is clear, any bottle entering the decollator via the side track may be decollated to the main track (recognition of a bottle at the decollator is performed by* Sensor B1*). Using* Sensor B2 *the leaving bottle releases the decollator for the next decollation.*

The realization of the junction consists of profile rails supporting the construction, several actors (transportation belt, decollator) and sensors (light barriers), wires and connections to a central communication bus, and software running on the controller of the transportation system. Interestingly, the specification mentions mostly actors and sensors, which act as a bridge between the mechanic and the electric/electronic domain. This is common for specifications of mechatronic systems, as actors and sensors are the most visible parts used to fulfill the system's tasks.

[1] As the specification is written in German we provide a translation here.

There are two links between the actors and sensors of a system. One is implicit by knowledge of physical laws and facts about the environment. For example, the piece of specification assumes that a bottle released from the decollator will appear at sensor B2 after some time (if the belt is turned on). This only holds as the laws of motion apply and if no one manually removes the bottle. The other link is realized explicitly in the system's internal logic, usually in software[2]. An example from the specification would be the sentence *"if [the] track is clear, [the] bottle [...] may be decollated [...]"*, which indicates an internally realized causal dependency.

To understand how the junction works, both links are required besides the mere knowledge of actors, sensors, and their positions. While the drawing from Figure 1.1(c) only contains the sensor/actor view, the program listed in Figure 1.2 is limited to the internal dependency between sensors and actors. The code also contains additional details, such as the fact that for B3 the value *true* indicates the absence of parts (line 2), while for B2 *false* has this meaning (line 10), for which there might be technical reasons. However, the logical conditions in the code can not be understood completely and checked for correctness without the spatial information on the positions of the sensors provided by Figure 1.1(c). Environment assumptions, which capture the external dependencies, are even only mentioned implicitly in the text of the specification. Thus, none of those views alone allows a complete understanding or analysis of this junction, and even in combination, the exact interplay between these views is left vague. This lack of a common integrated model affects both the early and late stages during development, as planning and exploration of design alternatives, as well as testing and virtual commissioning activities require knowledge of the implicit and explicit causal chains in the system.

1.2 Background

The motivational example demonstrates that for understanding and reasoning about mechatronic systems, a single integrated view of the mechanical, electrical, and software aspects is needed, as only the combined view of hardware and software behavior describes the functionality of the machine adequately.

1.2.1 Space-Intensive Mechatronic Systems

Mechatronics is a term coined in 1969 by Tetsuro Mori, derived from the words *mechanics* and *electronics*. Thus a *mechatronic system* is a system which provides its functions by an interaction of mechanic and electric effects. Today, the meaning is usually extended to include controller and software parts of the system, which were often subsumed under electrics when the term first

[2] It is also possible to realize complex sensor actor interactions with electric solutions (relays) or purely mechanical (*e.g.*, the purely mechanical calculators developed in the 17th century). However, as software realizations are most common today, we will use the term software even in the rare cases of purely electro-mechanical implementations of logic.

appeared. In this thesis we will use the convention, that a mechatronic system is a system in which mechanic, electric, and software components interact to fulfill a certain task. In some definitions the subdomains involved in such a system are listed in more detail, *e.g.*, listing fluidics as a separate discipline. As these detailed subdomains can be subsumed in the three given before and the more detailed view does not contribute to the primary discussion of this thesis' topic, we stick to the three subdomain definition.

During the last decade, the new term *cyber-physical system* has also often been used for these kinds of systems. It emphasizes the mixture of virtual/computational and physical parts in a single system and the interaction between them. The origins of this term are in the areas of control and power network engineering. The distinction to mechatronic systems is not clear and still heavily debated, but the emphasis of cyber-physical systems seems to be more on loosely coupled elements, such as sensor networks. The term cyber-physical system is used predominantly in the USA, were the field is heavily influenced by an NSF[3] research initiative with the same title, while in Europe and Japan, where the field is still dominated by mechanical engineers, the term of mechatronic systems is mostly used. In this thesis we will use the term mechatronics, but this could as well be seen as work in the area of cyber-physical systems.

There is a class of systems, where the spatial relationship between parts of the system and between the system and its environment is of primary concern. We will denote them as *space-intensive mechatronic systems* throughout this thesis. This class of systems covers besides others the entire range of factory automation systems and machine tools, which will serve as our main example. These systems typically perform multiple transformation steps on a physical product, thus the focus is on transportation and grouping of material, dealing with congestion, and avoiding collisions between multiple parts of the machine. All of these aspects are mostly determined by spatial properties.

Of course space and distances are also relevant for other domains commonly cited in the context of mechatronic systems. For example in automotive systems the distance to other cars is highly relevant, as is the distance to the ground for most avionic systems. However, usually the focus when observing these systems is on driving dynamics[4] or aerodynamics when studying the entire system, or on laws of combustion when focussing on the engines. So, we would count these systems as space-intensive only if the system is observed from a mostly spatial perspective.

1.2.2 Integrated Behavior Modeling

To understand the internal processes of a mechatronic system, it is not sufficient to treat the mechanical, electrical, and software parts in isolation, as the functions in such a system are usually realized by concerted interaction of all these disciplines. Instead the *behavior* can only be explained

[3] National Science Foundation

[4] While driving dynamics can be associated with spatial properties, as the position of the car changes over time, often the view is limited to acceleration and friction, and not to interaction with other objects in space.

by an integrated view of the system covering and connecting these subdomains. Interestingly, many aspects which are traditionally studied in single domain models, such as static (mechanical) stability, vibration characteristics, or power consumption, require an integrated inter-disciplinary view as well, when observed in more details. This is rooted in the fact that many chains of actions which were purely electric or mechanical have been partially realized by software controllers. Mechanical stability now depends on the way the (software controlled) actuators are moved, vibration is damped by countermeasures triggered by software, and power consumption can be affected by power saving schemes, which are – again – mostly realized in software.

The deeper understanding of a mechatronic system from the integrated behavioral view is needed in many phases during the development and life-time of these systems. Already in the requirements phase functional properties (small aspects of behavior) are usually not formulated with respect to one of the subdomains but with respect to the complete system. Often it is not even possible to break requirements down to subdomain requirements early on, as it might not be clear before the design phase which parts of the system's functionality are realized by some clever interaction of mechanics and electronics or rather mostly in software. The problem does not become easier, when it goes down to testing or formal verification, as the output signals generated by controller software are hardly interpretable without a deep understanding of the complete system. Finally, the integrated view is also the basis for documentation and discussion during the design and realization of a system, when engineers from multiple disciplines are involved. Instead of enforcing mechanical engineers to understand the software perspective, or vice versa, the integrated view can provide a common language.

Usually the discipline specific views on the system are already complex, but when trying to understand the system in its entirety from a detailed integrated view, this will be nearly impossible for most realistic systems. The key concept required to make this a manageable task is *abstraction*, *i.e.*, reducing complexity by focussing on certain aspects of the problem while neglecting others. This results in a *model* of the system. In general a model is an abstract view of an original fact which is created for a certain purpose. The goal of modeling is to better understand a certain aspect of the original by concentrating on certain properties and neglecting all others. If a model is backed by a formal modeling theory, which defines both the structure (abstract syntax, meta-model) and meaning (semantics) of valid models, we call it a *formal model*. So, the proposed solution to understanding a mechatronic system is to create an abstract (formal) integrated behavior model.

1.3 Problem Statement

In the previous section, we argued that integrated behavior models are crucial for understanding and developing mechatronic systems. There is, however, a lack of suitable modeling techniques and theories for these systems, as reported by Wilhelm Schäfer and Heike Wehrheim in [SW07]. They also mention that "as the component definition must include hardware and software components, the modeling language becomes domain specific at least to a certain extent". In this thesis, we

focus on the domain of factory automation systems, or more generally spoken on space-intensive mechatronic systems. An extension to more general mechatronic systems is briefly discussed in Section 9.4.

1.3.1 Lack of Practical Models

In the development of factory automation systems, the separation of the disciplines of mechanics, electrics, and software is still very common. Typically, mechanical engineering takes the lead by providing a fully detailed specification of the mechanic components of the system. This specification is then complemented by electrical engineers, and finally the software is written to work with the already existing electro-mechanical design [SW07]. However, such a sequential development process is often perceived as problematic as flaws in the design found during the software development phase (such as a missing sensor) require an additional expensive iteration of the entire process.

Additionally, the limitation to the discipline oriented views often leads too significantly less efficient and more expensive design solutions. The nature of mechatronics often allows to shift the realization of a function from one discipline to another one. If one of the disciplines is leading, this may result in a non-optimal system design as the strengths of the other disciplines are neglected. For example, the software might become substantially less complex and thus save resources in implementation and testing if an additional cheap sensor is introduced. However, if this is encountered late in the development process, changing the design is only possible at high cost, if at all.

One of the main reasons to still stick to the separation of disciplines is a lack of a common vocabulary between engineers of the different *faculties*. An integrated behavior model could fill this gap and act as a central development artifact. Especially the early planning phase, where we deal with coarse-grained models of mechanical, electrical and software aspects, benefits from such a model, as early integration of the disciplines helps to foster a common understanding of the system and can be used to explore design alternatives.

Furthermore, such an abstract model could be used for testing and virtual commissioning. Today it is still common that the automation system is not even tested before the regular commissioning phase, where a comprehensive system test is usually not possible, since the work takes place under high time pressure. Above all, failure scenarios and erratic behavior are only examined rudimentary due to the lack of time, missing specification of possible errors, and the risk of damaging the machine. Thus, errors are not identified and may appear during the operation phase. An integrated behavior model can be used to perform initial tests with a simulation of the system and perform test planning. With only little additional information the model could also be used for virtual commissioning, which is a hardware-in-the-loop test of the system's software using simulated machine parts. With appropriate techniques such a model could also be the base for the automatic generation of test cases or even for performing formal verification.

All of this depends on the availability of a suitable modeling technique, which should be built upon a solid semantic basis. This fosters a common interpretation of the model by everyone and allows the application of the advanced analysis and verification techniques described so far. Existing integrated modeling techniques, however, either fail to capture the behavioral aspects of the system, which are usually the most important ones from a software engineer's perspective, or lack a formal semantic foundation (*c.f.*, Chapter 3). This lack of a suitable modeling technique and corresponding tool support is perceived as a major obstacle for keeping the development of high quality systems a manageable task at ever increasing complexity.

1.3.2 Lack of Formal Models

While there is a plethora of modeling theories for pure software systems and also the engineering disciplines of mechanics and electrics provide many techniques for describing their view of a system, there exists no integrated modeling theory for the behavior of space-intensive mechatronic systems with clearly defined semantics and a solid mathematical foundation (*c.f.*, Chapter 3). We consider this lack problematic from both an academic and a practical perspective.

From the academic point of view the existence of a yet unsolved modeling problem alone can be seen as a worthwhile challenge. However, the lack of a modeling theory for space-intensive mechatronic systems also hampers research in many areas as new approaches are consequently limited to a single-discipline view of the system and thus are not able to exploit the full potential of mechatronics, *i.e.*, the fine-grained interaction of mechanics, electrics, and software. Examples are analysis techniques for predicting wear or energy consumption. Today these analyses are typically limited to the mechanical view and thus have to calculate with a maximal or average stress on individual actors (drives, etc.). More precise predictions are possible if the internal machine behavior (mostly described by the software) is taken into account, as interdependencies between actors then become visible, which affect the overall wear and energy consumption. For example the machine's internal logic could ensure that two drives are never running at the same time, thus avoiding certain vibrations in the machine during operation.

More crucial is the effect in practice, as we feel that many of the shortcomings of existing modeling tools are caused by the lack of an underlying modeling theory. One of the most striking examples is found in the manual of the simulation tool *Sinumerik Machine Simulator*, developed by Siemens for the creation of virtual commissioning models. The models used there are based on a data-flow graph. The manual [Sie03] of version 5 states that in case of cycles the execution order is undefined and one of the components (called equations there) will use an outdated input value as input[5]. While nondeterminism is a common tool in modeling to express uncertainty or capture the degrees of freedom for implementations, here it seems that the uncertainty is caused by a lack of

[5] Original text is in German: *Durch die Rückkopplungen wird zwangsweise immer eine der [...] Verknüpfungen mit einem Eingangswert rechnen, der aus dem vorherigen Rechenzyklus stammt. In solchen Fällen ist nicht festgelegt, in welcher Reihenfolge die Gleichungen bearbeitet werden.* Although the manual was written in 2003, the same version of the manual is still used for the current release of the software.

understanding of the theory of data-flow networks. This is especially irritating as loops in data-flow networks are well-understood in theory (*c.f.*, Section 5.1.3). The theoretic flaw is problematic in practice, as the modeler has no way to resolve the nondeterminism and a limitation to loop-free data-flow models would severely impact the expressiveness of the models.

1.4 Contribution

This thesis contributes to both the academic and the practical aspects of the mentioned problems. We introduce a semantically founded modeling theory for space-intensive mechatronic systems by extending the ideas of stream-based interface descriptions as introduced in [BS01]. The model captures the positions of objects over time and supports basic concepts required for the modeling of realistic systems, such as the collision between solid objects, the detection of objects in certain locations, material flow, and kinematic relationships between objects of the model. At the same time primitives for expressing computation and communication are available. We describe a set of operations, which can be used to systematically build spatio-temporal models from smaller parts and thus support system decomposition and reuse on the model level. We thus extend the body of modeling theories by a suitable model for space-intensive mechatronic systems, which we perceive as a first step towards a comprehensive model of general mechatronic systems.

Based on the mathematical modeling theory a practically applicable integrated behavior model is described. This model can act as a link between engineering departments and captures all aspects of a machine (mechanics, electrics, controller), albeit in an abstract and simplified manner. Thus, it allows exploration of design ideas and simplifies discussions with a customer. Refining this model by adding more details allows to use it as a basis for discipline-specific models (*e.g.*, CAD models). By using the model to explore design alternatives, synergies emerging from mechatronics can be exploited and the overall development process gains flexibility. Based on the operationalized model, we implemented a prototypical tool that realizes many of the functions needed to support various activities in a model-driven mechatronic development process. This includes early validation through simulation, tracing and consistency checking to other engineering models, and generation of simulation models for virtual commissioning. The applicability of both the model and the tool to real-world factory automation systems is evaluated in three case studies.

Finally, this thesis provides an example for the integrated development of modeling theory and modeling tool. Typically, either of both is of limited use in isolation, as a modeling theory without practical realization can not be applied to real-world problems, while a modeling tool without an underlying theory often suffers from inconsistencies in the interpretation of the model and limitations in the expressiveness due to a choice of inappropriate abstractions. Interlocking the development of theory and tool helps to improve both the academic and practical results. Building a practical model and a tool based on a clean theory significantly reduces efforts due to a well-understood and described basis. Contrary, the modeling theory benefits from insights gained during the integration of the theory into a tool and from its practical application.

1.5 Contents

Chapter 2 presents preliminary definitions and thoughts that are relevant for a better understanding of the ideas behind the model discussed in this thesis. Chapter 3 summarizes related work and explains the differences to other approaches and areas of research. This underpins the lack of suitable models as argued before. Chapter 4 lays the foundation for the modeling theory by summarizing mathematical preliminaries and discussing different notions of space and time. There also the connection of space and time in terms of spatio-temporal models is briefly discussed. The theoretical model is introduced in Chapter 5. Based on the FOCUS theory a basic stream-based model of spatio-temporal components is presented, which is then extended to support the description of the material flow. Based on the modeling theory an operationalized model is developed in Chapter 6. It explains the simplifications necessary to obtain a practically applicable meta-model and is thus the first step towards tool support. The actual tool as well as decisions taken and lessons learned during its implementation are summarized in Chapter 7. As the implementation itself is only of minor interest, we concentrate on features of the tool which support the mechatronic development process, such as support for component libraries, tracing to technical system models (X-CAD), and virtual commissioning. We report on the application of the model and tool in the context of two industrial case studies and one non-industrial model in Chapter 8. This evaluates our approach and demonstrates its applicability to real-world systems from the automation domain. Finally, Chapter 9 concludes by summarizing the results and outlining open topics and future work. While the work is presented in the classical top-down fashion for the sake of simplicity, this of course does not reflect the actual development, which took place in multiple iterations over the course of several years and four industrial and government funded projects.

Previously Published Material The material covered in this thesis is based in part on contributions in [HB08, BH09, Hum09, HH09, BHL09, BHH+09, BHLH09, BHH+10].

2 Preliminary Considerations

This chapter discusses concepts which are fundamental for the approach and model presented in this thesis. We start by recapitulating the abstraction layers used during systems development and discuss how our approach fits into these layers. After this, we describe the role of space in both the automation domain and our model. This also deals with the question why we chose to focus on space. The following section explains why and under which assumptions existing behavior models are ill-suited for the description of space-intensive systems. Finally, we derive from the identified limitations a set of requirements that are used both for the development of the modeling theory presented later and the classification of related work.

2.1 The Role of Abstraction

The most important aspect about modeling is *abstraction*, *i.e.*, the reduction of a system to the properties which are required for a certain purpose. This helps to keep the modeling effort low and also to better understand an aspect of a system by concentrating on specific properties and neglecting all others. In this section we briefly introduce abstraction layers proposed for automotive development and discuss their applicability to the automation domain.

Abstraction Layers A common feature of system development processes is that the system is described by different models (or views) over the time of development. These models are enriched with details during development and reside at different levels of abstraction. Here we follow the approach from [BFG+08], which describes a model consisting of three abstraction layers. While the model targets the automotive domain, it is general enough to be applied to other domains as well. The three abstraction layers are as follows.

- The **usage layer** defines the functions of the system. It answers the question, *what* the system shall do. Typical description techniques comprise function or service hierarchies and textual requirement documents.
- The **logical layer** describes a solution or implementation of the functions defined in the usage layer. The focus is on the underlying solution idea and not the technical realization. This layer explains *how* the functions of the system are realized. It can be described using, for example, component or automaton based description techniques.

- The **technical layer** maps the logical model to a concrete realization. It describes the technical/physical components in detail and thus describes *by which means* the functions are provided. At this layer we can find, for example, bus topologies, CAD models, or deployment diagrams.

Application to Automation Engineering The presented abstraction layers are also well suited for development in the automation domain. In practice, the usage layer is typically described by textual requirements documents while the technical layer consists of various CAD models (for mechanics, electrics, fluidics) and part lists. The logical layer is often omitted or only described very vague by path-time diagrams. This causes two major problems. One is that the step from the usage to the technical layer is rather big without an intermediate representation and thus requires high effort. This results in a big-bang approach which leaves little space for exploration and experimentation and has to be right at the first time, as otherwise many costly iterations are required. Models at the logical layer improve this situation by allowing iterations at a more abstract level, which can be performed much more efficiently in terms of engineering resources. The second problem is the lack of a model that integrates the views of the different disciplines. The requirements on the usage layer are either too general, or specific to the disciplines, while the technical (CAD) models only capture a specific view of the system (such as mechanics) and are usually too detailed to be integrated. Consequently, integration happens not before assembly and commissioning – a point where changes to the system are increasingly expensive. An obvious candidate for the integration of the disciplines are the models of the logical layer.

This problem of missing models at the logical layer for the development of automation systems is one of the motivating factors for the work of this thesis. The goal is to find a model which is sufficiently abstract to be used at the logical layer and at the same time integrates the views of the different disciplines. As explained in the next section, spatial properties are very important in the automation domain. Thus, the logical layer should allow to capture them.

2.2 The Role of Space

In this section we discuss the role of space in the automation domain and argue why the focus of this thesis is on the inclusion of spatial properties. In addition, we briefly discuss the relationship of spatial properties to other system properties, both physical ones (such as temperature) and quality requirements (such as safety). Finally, we bridge the gap to the previous section by discussing abstraction and space.

Space in Automation Engineering Systems in automation engineering are centered around production processes, which transport, assemble, modify, or somehow manipulate material. Many of the effects that can be observed in these systems are of spatial nature. Examples are material

transport (*i.e.*, change in position), congestion in conveyors (caused by spatial collision), activation of photo-electric barriers (spatial occlusion), assembly of parts (*i.e.*, creating a certain spatial relationship between sub-parts), or grinding processes (change of spatial shape). In an unpublished case study, the *iwb*[1] found that up to 80% of the modeling efforts for a simulation model used in hardware-in-the-loop tests went into the description of spatial properties and the reactions to changes in them. So, while other properties (such as temperature) may be important as well, the spatial aspects seem to be dominating in the automation domain.

The motivation for concentrating on space when working on behavior models for mechatronic systems is twofold. First, creating a model that incorporates all possible mechatronic effects is likely to fail due to the complex interactions possible between all of them. Thus, we limit our scope to space as the property that has most impact. Second, space is a prerequisite for many other mechatronic effects. For example, to describe the propagation of electromagnetic waves, which may cause malfunctions in electronic devices, the spatial setup is important. As such, the integration of space into behavior models can be seen as a preparatory step for the inclusion of other aspects of mechatronic systems. Consequently, we concentrate on space in this thesis but also discuss the inclusion of other aspects briefly in Section 9.4.

Spatial Requirements Using a model for specification purposes is a way of formalizing system requirements. While a behavior model typically captures functional requirements, the non-functional ones are usually organized in different kinds of models (such as [Dro95, DWP+07]). As the focus of this thesis are behavior models, we should elaborate on the question whether spatial properties are functional or non-functional, to better understand the suitability of extending the behavior model by spatial aspects.

The differentiation between functional and non-functional requirements is a heavily discussed topic in requirements engineering, and there is yet no agreement on clear definitions that provide solid criteria for both. The definitions in software engineering text books [Som06, PA10, BD04] are typically along the lines that *functional* requirements specify functions that the system should support, *i.e.*, output reactions to certain stimuli, while *non-functional* requirements are everything else. As Martin Glinz observes in [Gli07], also "the notion of non-functional requirements is *representation-dependent*". The example there is that a requirement "the system shall prevent any unauthorized access to the customer data" would be considered non-functional, while the more concrete "the database shall grant access to the customer data only to those users that have been authorized by their user name and password" is a (security related) functional requirement. The situation is similar for requirements based on physical effects found in mechatronic systems. For example, a requirement "the overall system temperature shall not exceed 60°C to avoid overheating" would be considered non-functional by almost all definitions. Contrary, the requirement "the tip of the welding torch must reach a temperature of 1600°C to melt steel", which is also temperature related,

[1] Institute for Machine Tools and Industrial Management of Technische Universität München

would usually be categorized functional, as the melting of steel is part of the function of a welding system.

Following [Gli07] we think that we can not categorize entire classes of requirements (such as security or temperature related requirements) as functional or non-functional. Instead, it depends on the concrete characteristics of a particular requirement. Consequently, spatial properties are not *per se* functional or non-functional, but they exist in both flavors. Examples of functional requirements would be "as soon as the sensor detects the presence of a bottle, is it capped", or "the robot picks every 10th brick and places it at the measurement station". Non-functional requirements related to space are "the entire system has to fit into a 10 to 20 meter factory building" or "there may be no sharp edges protruding into the maintenance area". As we deal with behavior models, our focus is clearly on the functional spatial requirements. Still, some of the non-functional ones can be expressed using our model.

Abstraction and Space The amount of spacial information contained in a model depends on the modeling purpose. Engineering practice can lead to the impression that geometry has to be either omitted or described down to the last screw. However, the goal of abstraction is a reduction to those properties that are relevant for the task at hand. The task at the logical layer is the description (and possibly simulation) of the underlying behavior of the system. So we are not interested in the geometry of all the screws and gears built into the machine, but rather limit our view to the rough shape of functional elements, such as robots or belt conveyors. However, if a certain gear or screw is essential to supply a function of the system, it has to be modeled with as much detail as necessary. For example, if some gears are in an area where a robot could collide with them and the avoidance of such a collision is essential, the gears should be modeled. But it is sufficient to approximate the gears by a rough shape which represents a *safety area* around the gears instead of modeling all teeth of the gears and their exact rotation over time. Similarly, a belt conveyor can be described by a simple delay element where objects entering on one side will exit on the other side after a certain time (depending on the belt's speed). For such a description no geometry is required. However, if the position of the objects on the belt is relevant for the functioning of the system, it should be captured even at rather abstract modeling levels. For example, the conveyor could be part of a complicated transportation system, where collisions at junctions are to be avoided and robots are in place to pick objects from the belt. In addition to functional reasons, including spatial details even in abstract models can sometimes help to better understand the model as especially mechanical engineers are trained at thinking in geometric terms.

2.3 Problems of Behavior Models with Space

One of the first questions that should be answered when proposing another modeling language is why not one of the existing techniques could be applied to the underlying problem. This section

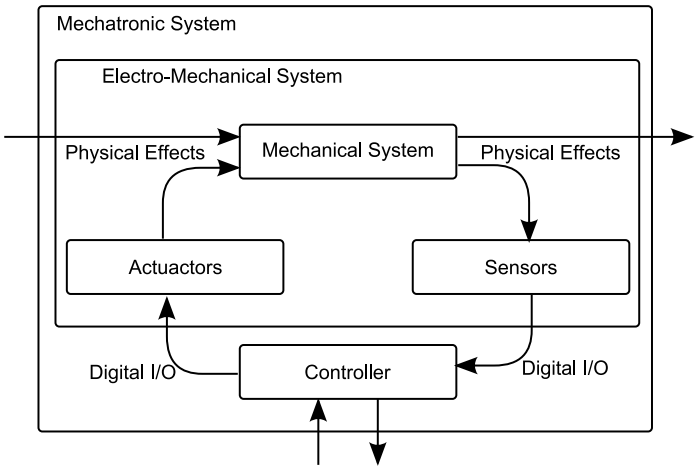


Figure 2.1: One possible view of a mechatronic system

attempts to give the answer to this question by describing the problems when using existing behavior models known from modeling software systems.

A typical reason for modeling the behavior of a mechatronic system is to provide an environment model for an embedded controller. In this case, the primary goal is to create a model of the controller, but to analyze and simulate it, the environment of the controller has to be described formally. For such a task, often a view of the system as provided by Figure 2.1 is used, where actuators and sensors are mediators between the digital/logical and the mechanical/physical world. We will refer to the mechanical system in conjunction with the sensors and actuator as the *electro-mechanical system*, while we refer to the entire system as the *mechatronic system*.

In the remainder of this section we first explain, what a solution using the existing *well-known* behavior models could look like. Based on this we present the various problems with such a solution and conclude with a discussion of the consequences of these problems. The focus of this section is on behavior models as they are known from software modeling. Models from mechanical engineering are usually either related to them, or do not deal with behavior over time at all. An overview of modeling techniques is also given in Chapter 3.

Apparent Solution Given the previous setup, modeling the behavior of the controller is well understood in software engineering and there are many formalisms, such as state charts or FOCUS (see Section 3.1 for more details and other models), which can be used to describe a model of the controller. While of course the various models differ in details, such as the timing model and the notation used, the model of the controller can be boiled down to a relation between its digital inputs and outputs. This relation is then usually decomposed into smaller manageable parts which are described using one of various techniques.

What we are interested in here, is the modeling of the environment of the controller, which is given by the electro-mechanical system (we will neglect any additional input from the user or connected systems in this section). A key observation is that from the controller's perspective, the electro-mechanical system only communicates via the digital inputs and outputs, thus it can also be interpreted as a relation between them. Consequently, we can directly capture it as a FOCUS component or a (possibly infinite) state machine[2]. However, the attempt to directly model the electro-mechanical system using one of these formalisms has severe limitations, which are explained next.

Problems with Modeling The previous paragraph states that in fact the electro-mechanical system (and thus the entire mechatronic system) can be modeled using one of the well-known behavior models. However, it does not give any hint of how easy or complicated it is to describe it using one of these formalisms. In fact it turns out that modeling is getting more involved as soon as spatial properties are relevant for the systems. Typical examples for such properties from the automation domain include the activation of photo-electric barriers by other mechanical parts (leading to a signal to the controller via a sensor) or a collision between mechanical parts that blocks their motion and thus delays or hinders later sensor activations. Often these properties can not be ignored during abstraction as they are central for the behavior of the system. Unless the geometry is simple (for instance aligned on a grid) or contributes only a minor part of the system, all the rules for transformation and collision testing of non-trivial geometry have to be encoded by the engineer using these models. Unless this is supported by the modeling language, this is a tedious process.

Problems with Composition Another problem of typical behavior models in the context of space-intensive systems is composition. To manage the complexity of the models, nearly all approaches support a decomposition into separate interacting submodels, often called *components*. Typically, this interaction is also limited to further structure the model and ease analyzability. For example, in FOCUS only components connected by channels can exchange messages. Many spatial properties in contrast are of a global nature, *e.g.*, all robots in a system can in principle activate all proximity sensors within their respective range. To model such a situation we either require some kind of broadcast communication or explicitly connect each robot component to each sensor component. The first solution again makes modeling complicated as all components now have to include logics for dealing with these broadcast messages (if the model supports them at all) and filter the relevant ones. The second solution involves a lot of effort when adding new components, as they have to be connected to most of the existing ones.

Problems with Reuse An important ingredient of a modeling technique to support engineering is reuse of components. A robot, for example, should be modeled only once and then be used

[2] A different argument is along the line, that simulation models for the electro-mechanical system exist, which are used in parallel to a controller implementation during virtual commissioning. As these models are running on discrete hardware, ultimately the entire simulation and hence a model of the system can be described by a discrete automaton.

multiple times. When using a plain behavior model where support for collisions is implemented by the user in terms of the modeling language, reuse becomes more difficult. The reason is that the spatial position of the robot in the system affects its interaction with other components, as collisions are only possible with nearby parts. Thus, the robot's position must be made a parameter of the component and has to be respected in the user-contributed collision calculation, which requires additional modeling effort. In contrast, with an explicit model of space the position is a property of the component and can be easily accessed and changed.

Problems with Material Flow Material and its movement and transformation in the system is often an important part of the behavior of the electro-mechanical system (at least in the automation domain). Usually the material travels through a large portion of the system and thus interacts with most of the components. Using typical behavior models, a possible solution is to only respect material within each component, *i.e.*, in each component manage the material currently *owned* by it. Passing material between components can be expressed by special messages that encode the position and properties of a piece of material. The problem with this approach is that the *ownership* of material reduces decomposability as all system parts that can interact with a material piece *at the same time* have to be modeled by a single component. In addition ownership of a material is not always clear, as a piece of material can be easily affected by multiple components, such as on the boundary between two conveyor belts.

Conclusion The previous paragraphs explained different problems that occur when modeling the electro-mechanical part of a mechatronic system using behavior models targeting pure software systems. Of course all of these problems depend on the level of abstraction used. If the respective details can be omitted, the model becomes significantly easier. For space-intensive systems, as often found in the automation domain, usually the spatial properties can not be neglected as they significantly contribute to the behavior. This leads to a severe overhead when modeling them with these modeling techniques.

Our contribution to this problem is to support direct encoding of spatial properties as part of our modeling theory. Spatial objects are first class elements and collision is a part of the model's semantics. Material properties and material flow can be expressed explicitly, too. This unburdens the modeler from encoding spatial aspects indirectly using other language constructs and thus simplifies the creation, understanding, and analysis of models.

2.4 Requirements to a Suitable Modeling Technique

Based on the previous discussion and experience from four projects in the area of modeling and virtual commissioning for automation systems, we derived a list of requirements, which were used

as a guide during the development of our modeling approach for space-intensive mechatronic systems. These properties will also be used to assess existing and related work in the area of modeling systems in Chapter 3.

The first three requirements listed are applicable to general system models and reflect our understanding of fundamental ingredients to any modeling technique. The second set consists of properties that should be easily expressible by the modeling approach. This list of properties is based on our focus on space-intensive systems and the domain of factory automation systems. As described in the previous section, all of these properties can be *simulated* by any modeling technique which is sufficiently expressive, *e.g.*, by use of special variables and subroutines or the correspondence in the respective behavior model. However, as all of the properties listed are common and, thus, frequent in systems from the automation domain, treatment of the those aspects as *first class entities* greatly simplifies the creation and understanding of models. Additionally, only by handling those properties explicitly, can we explicitly respect them in modeling and composition operations and provide specific techniques for analysis of these models.

Clearly Defined Semantics A modeling technique for mechatronic systems should be based on a formal modeling theory that ensures semantic soundness and allows the formal analysis of system models. The existence of clearly defined semantics ensures that the meaning of the individual modeling elements and especially their combination is well-defined. Tools built upon models without explicit semantic foundation (*i.e.*, where the semantics is defined only *by the implementation* of the tool) often suffer from subtle problems in non-trivial combinations of modeling elements (an example has been given in Section 1.3.2). This leads to situations where a user of the tool can not easily predict the meaning of certain models but only explore it by *trial and error*. Additionally, without semantic foundation different implementations of the modeling tool (maybe for different tasks such as simulation and verification) may interpret the meaning of a specific model differently.

Different Levels of Abstraction One of the goals of modeling is the reduction of a system to certain aspects, also known as abstraction. The two main reasons to use abstraction is to make modeling easier and faster as irrelevant details can just be omitted, and to focus on certain parts or properties of a system by neglecting those we are currently not interested in. Different levels of abstractions can also be combined in the same model. For example a certain subsystem could be modeled with more details, while the surrounding subsystems are only modeled detailed enough to capture their interaction with the more detailed subsystem. A modeling technique should not by itself enforce a single level of detail or abstraction, but allow the modeler to choose one. Ideally, it should support the combination of model parts with different abstraction levels in the same model as well.

Notion of Interfaces and Composition Composition is the process of building larger models from multiple smaller ones. This helps in building more complex models by *decomposing* them into a hierarchy of smaller models which are then composed to form the complete system model. Composition is also a prerequisite for reusing model parts, which are often referred to as *components*. Reuse is important especially for mechatronic systems, as subsystems, such as sensors or belt conveyors, often appear multiple times within the same system. While composition obviously is a prerequisite for reusable components, we also require a notion of (component) *interfaces* from a modeling technique. The interface describes the possible interactions with the component and supports encapsulation as the internal implementation (or modeling) details are hidden. It is used as a more compact description as it is sufficient to understand the interface and not the component's implementation, it can be used to decide on substitutability of components with the same interface but different (internal) realization, and it allows to decide whether two components can be combined at all (*composability*). Any practically applicable model should support a notion of components with clearly defined interfaces and composition operators.

Spatial Relationship between System's Parts The shape and relative position of most of the physical parts of the system has a major impact on the system's behavior. Subtle changes to these spatial relationships can completely change the functions performed by the system. Thus, the model should directly capture geometric information or at least an abstraction of the system's geometry that captures the spatial properties well enough. As the model is an abstract view of the machine, more abstract geometry should be sufficient for most purposes. A benefit of directly including geometry is also the possible support for visualization of the model, which is a key requirement for making the model understandable to non-experts.

Material and Material Flow The primary purpose of production machines and factory automation systems is the processing of material, *i.e.*, its transportation, (dis)assembly, and transformation. What makes the modeling of material complicated is that it is not associated with a fixed part of the system but rather runs through the entire machine. This is often also referred to as the system's *material flow*. Furthermore, material often is relevant to the system only during a finite time span between insertion into the system and either consumption or extraction. Thus, the simplification of a system running for an infinite time, which is often applied in systems modeling, does not easily apply to the material part of the model. Thus, the modeling technique has to account for material and its interaction with the remaining model of the machine.

Permanent and Temporary Kinematic Relationship While the spatial properties capture shapes and positions, *kinematic* describes the *possible* relative movements of two or more parts. It can be viewed as an extension of spatial aspects over time. As we also deal with material, not only permanent kinematic links (like for the segments of a robotic arm), but also temporary relationships (like a bottle being gripped and thus following the robotic arm, but following the motion of a transportation belt after being released) have to be supported in the model. Many models in the area of

physics simulation do not express kinematic relationships directly, but rather *implicitly* by describing joints (or more generally spatial constraints) from which the kinematics can be calculated if the forces involved are known. Contrary, we prefer an *explicit* model of kinematic relationships, as our focus for modeling is more in the planning phase, were the intent of the modeler should be captured, while the implicit model already anticipates a realization in terms of joints. Additionally, explicit relationships can simplify the analysis of the model, especially chains of motion.

Expression of Communication and Energy Transmission Theories of systems often differentiate between three flows exchanged between system parts and the system and its environment: information, energy, and matter. In modeling, typically the intent of the flow is expressed rather than the physical reality. For example on a communication bus also energy is exchanged, as electric current is used for sending the individual bits of a message on the wire. However, the primary intent is information exchange. Similarly, hydraulic fluid is clearly a matter flow, but it is usually used to transmit energy[3]. In our requirements, transfer of matter is already handled in part by the material flow, but also communication and energy transmission need to be supported. While communication is a common part of many behavior models, energy transmission is not. Although in many cases energy transmission can be modeled approximately as communication, in some cases this is not appropriate or at least cumbersome. The model should be able to explicitly express all three kinds of flows described here.

2.5 Summary

This chapter introduced a model of abstraction layers used for systems development and explained how our approach fits into these layers. In this context, we also discussed the relation of abstraction and spatial properties. Then we discussed the weaknesses of existing approaches when spatial properties and material flow are important aspects of a system. This implies that a modeling technique with explicit support of space and material is more appropriate for describing these systems. Based on the problems described and our own experience, we developed a list of requirements for such a modeling technique. These requirements are also used to categorize and assess the related work in the next chapter.

[3] There are also cases, where the amount of hydraulic oil exchanged is relevant. An example would be to include the possibility of leaks in the hydraulic pipes. In such a case the model should capture both the flows of energy and matter.

3 Related Work

This chapter summarizes existing work that is similar or otherwise related to the topic of integrated behavior modeling in the area of space-intensive mechatronic systems. As the body of potentially relevant work is large, only the most prominent of several related or similar approaches is presented. Especially in mechanical engineering commercial tools have a serious impact on system modeling, thus not only academic work but also these tools are considered here.

For each approach we provide a short summary and describe the perceived obstacles to its application to the behavior modeling of space-intensive mechatronic systems. To simplify the comparison of all approaches, the same criteria are used to assess them, which are based on the requirements from Section 2.4. We describe the assessment results by an ordinal scale using the set $\{\text{-}, \text{o}, +\}$. The results will be given as a table for each approach described and explained in the text if necessary. Of course no set of criteria can faithfully capture all details of the approaches, so we complement them in the text where considered necessary. The following sections group the related work into common categories. Approaches which could be discussed in multiple of these sections are placed in the section we consider most appropriate. Finally, the approaches' classification according to these criteria is summarized in Section 3.7.

3.1 Formal Models of Software

There is a wealth of techniques used for formally modeling (embedded) software systems. While they have usually never been designed to also capture the non-software aspects of a system, we include them here as many of them are used as a basis of or at least heavily influenced models for mechatronic systems.

Statecharts Statecharts are a special kind of hierarchical automata introduced by David Harel [Har87]. Statecharts allow control states to contain substates which are either XOR states (corresponding to states in classic deterministic automata, where exactly one state is active at one time) or AND states which may be active at the same time. These AND states are used to model orthogonal behavior and can be interpreted as parallel independent computation. Transitions may connect not only

Semantics:	+
Abstraction:	+
Interfaces & Composition:	-
Spatial Relation:	-
Material & Material Flow:	-
Kinematics:	-
Communication & Energy:	-

primitive states (states without substates) but also states in the hierarchy. This helps to avoid redundancy in the expression of transitions that affect entire sets of states, such as mode switches or error handling.

While the semantics of Statecharts seems simple at a first glance, the interaction of these basic elements can lead to subtle problems. A paper by von der Beeck [vdB94] summarizes 19 problems in the original sketch of the Statechart semantics presented in [Har87]. Depending on the solution chosen for these problems, different variants of Statecharts can be developed. The mentioned paper reports on 23 Statechart variants found in the literature, before introducing a 24th. Thus, there is a solid semantic foundation for Statecharts (after agreeing on one of the available semantics). In terms of abstraction, Statecharts can be used to model a system in varying levels of detail. However, Statecharts themselves provide no clean notion of an interface and thus do not support components which can be combined with each other and reused. The specific requirements of space-intensive mechatronic systems are not supported.

Input/Output Automata Nancy Lynch proposes input/output automata [Lyn03] as a model for distributed event systems. They are automata whose edges are labeled with *actions*, *i.e.*, they consist of a set of states, a set of initial states, a set of actions, and a transition relation mapping each state/action pair to a set of states. The set of actions is partitioned into input actions, output actions, and internal actions. A system is described by a set of such automata corresponding to individual components or subsystems of the entire system. The essence of composition is that transitions labeled with an input action π have to execute whenever another automaton takes a transition labeled with output action π. This corresponds to synchronous communication, as sending a message (output action) and receiving the message (input action) happen at the same time. To avoid blocking, input/output automata are required to be *input enabled*, *i.e.*, for each pair of a state and input action, the transition relation must define at least one successor state. The input/output automaton model provides a technique with clearly defined semantics and allows to model at different levels of abstraction. The formalism supports interfaces to be defined by sets of valid actions and supports composition of automata. Being an abstract model for discrete event systems, however, our domain specific requirements are not fulfilled.

Semantics:	+
Abstraction:	+
Interfaces & Composition:	+
Spatial Relation:	-
Material & Material Flow:	-
Kinematics:	-
Communication & Energy:	-

Petri Nets Petri nets were first described in Adam Petri's dissertation [Pet62]. They can be interpreted as a directed bipartite graph consisting of two kinds of nodes, called *places* and *transitions*. The state of a Petri net is described by an assignment of so called *tokens* to the places. A transition is called *active*, if all places connected to it by incoming arcs contain at least one token. An active transition may *fire*, which causes one token to be removed from each of the input places, and one token to be added to each place connected to the transition by an outgoing arc. A run of a Petri net consists

Semantics:	+
Abstraction:	+
Interfaces & Composition:	o
Spatial Relation:	o
Material & Material Flow:	o
Kinematics:	-
Communication & Energy:	o

of a sequence of fired transitions. Numerous extensions to the basic Petri net formalism have been proposed, including variants with fractional tokens, edge weights, or tokens of different type.

Some of these Petri net variants are also proposed as a modeling formalism for automation systems (*e.g.*, [Fer94, ZZP07]). In these models the tokens usually represent a tool used by the machine or a product (material) being processed. The places (at least some of them) then correspond to individual stations (subsystems) of the machine. This allows to capture a logical view of space, but is not capable of expressing actual geometric positions, which are required to model certain spatial relationships detailed enough for simulation and testing. While this allows to capture the material flow to some degree, kinematics is not easily expressed in these models. While the semantics for Petri nets are well studied and models of different degrees of abstraction can be described, the weak point of Petri net based approaches is their lack of clear interfaces and operations for composition, although some extensions of Petri nets do provide composability (see *e.g.*, [Kin95]).

Process Algebras A process algebra or process calculus describes a system by (concurrent) processes. These processes are described by (potentially recursive) equations including process variables, primitive processes called *actions*, and operators over processes. This formal framework allows an axiomatization of process equivalence and, hence, supports proving certain properties of processes. The three most prominent process calculi are Tony Hoare's *Communicating Sequential Processes (CSP)* [Hoa85], Robin Milner's *Calculus of Communicating Systems (CCS)* [Mil80]), and the *Algebra of Communicating Processes (ACP)* by Jan Bergstra and Jan Willem Klop [BK87]. Some other basic process algebras and many extensions of these initial algebras exist, including extensions for real-time and dynamic systems.

Semantics:	+
Abstraction:	+
Interfaces & Composition:	o
Spatial Relation:	-
Material & Material Flow:	-
Kinematics:	-
Communication & Energy:	-

Process algebras suffer from the same problems as other modeling approaches targeting software systems when applied to space-intensive mechatronic systems: a lack of support for the specific properties and problems of these systems. In addition, while having a strong semantic foundation and providing means of abstraction, most process algebras have a weak notion of interfaces (although being strong in terms of formal composition operators).

Lustre Lustre [HCRP91] is a representative of the synchronous dataflow languages and was developed by Nicolas Halbwachs et al. in the 1980s. It is based on the notion of *flows*, which capture the value of a variable over time sampled by a discrete *clock* associated with the flow. Flows can be defined and combined using several operators. The flows can be interpreted as connections between operators leading to a dataflow view. As Lustre is a synchronous language, communication (*i.e.*, data transfer) between these operators happens at the same time without any delay. Composition can be expressed by recursive equations on expressions of flows. These equations, however, have

Semantics:	+
Abstraction:	+
Interfaces & Composition:	+
Spatial Relation:	-
Material & Material Flow:	-
Kinematics:	-
Communication & Energy:	o

to be chosen carefully, as the synchronous nature of Lustre allows the formulation of systems of equations with no valid solution or multiple possible solutions. These so called causal loops are detected by special analysis techniques or compilers. There also is a tool realization of Lustre called SCADE [Ber07], which integrates a Statechart-like graphical notation with the data-flow notation of Lustre.

Just like the modeling languages described before, Lustre is strong in terms of semantics and abstraction and also supports composition and (data-flow) interfaces well. The requirements specific to our modeling problem, however, are not directly supported. Only the case of energy flow can be expressed to some degree by the use of flows.

Semantics:	+
Abstraction:	+
Interfaces & Composition:	+
Spatial Relation:	-
Material & Material Flow:	-
Kinematics:	-
Communication & Energy:	o

FOCUS FOCUS is a modeling formalism propagated by Manfred Broy [Bro08, BS01] based on streams and stream-processing functions. A stream is a finite or infinite sequence and is used to model the messages exchanged on communication channels over time (channel history). FOCUS clearly separates the syntactic and semantic interface. The syntactic interface describes the number, name, and type of incoming and outgoing channels (*i.e.*, the static aspects), while the semantic interface describes the component's reaction (outputs) to certain input sequences (*i.e.*, the dynamic view). The semantic interface is described by a strongly causal relation between the stream of the input and output channels. For the actual definition of the function, several specification styles are proposed. Its main difference to Lustre is the concentration on strong causality and thus an *asynchronous* model of communication and computation. This also simplifies composition, which (just as in Lustre) is expressed by systems of (potentially recursive) equations which can be interpreted as a connection between the component's channels. The strong causality ensures that such a system has exactly one solution and thus composition is always well defined. There also is a tool implementation of the FOCUS theory called AutoFOCUS [BHS99, SPHP02]. More details on FOCUS are presented in Section 5.1, the differences between asynchronous and synchronous languages and their consequences are discussed in Section 5.1.3. The strengths and weaknesses regarding our criteria are the same as those of Lustre.

Summary All of the models included in this section can to some extent be used for modeling spatio-temporal systems by encoding spatial properties by suitable data structures and treating them as data state. As this encoding is rather tedious and obstructs the view to the underlying problem, we do not consider this a solution to our modeling problem. Furthermore, the spatial properties (when represented by special data structures) are not respected during composition. This is not surprising, as these models were developed with mostly software systems in mind, often within an embedded context. In contrast, the formal semantics and the possibility for modeling at different levels of abstraction are provided by all of them and many support mechanisms for defining components with clear interfaces and a notion of composition.

The main strength of these formalisms is in the behavior description, so we conceive them as a solid basis for a model of space-intensive mechatronic systems. The work presented here can be seen as a spatio-temporal extension of FOCUS. The reason to use FOCUS as a basis was influenced by both the strong notion of interfaces and composition (which are beneficial when structuring a system and constructing reusable components) and the asynchronous model of computation (*c.f.*, Section 5.1.3).

3.2 Models for Hybrid Systems

Hybrid systems are systems with both discrete and continuous state changes[1]. There is a rich set of formalisms for the description and analysis of hybrid systems. Many of them are extensions of the software models described in Section 3.1. Examples of systems modeled in these formalisms include simple thermostats, where a (continuous) temperature is controlled by switching a heater on and off (discrete), or the railway crossing example consisting of a train (modeled by its continuous distance from the crossing), a gate, and a discrete controller which has to ensure that the gate is closed whenever the train is near the crossing. There are also more complex case studies described in the literature, such as that of a pitch controller for a helicopter [MWLF03].

Semantics:	+
Abstraction:	+
Interfaces & Composition:	-
Spatial Relation:	o
Material & Material Flow:	-
Kinematics:	o
Communication & Energy:	-

Hybrid Automata Hybrid automata, as described by Thomas Henzinger [Hen00], are probably the most well-known model of hybrid systems. They are described by a set of real-valued variables (interpreted as functions over time) and a control graph, *i.e.*, control states connected with directed transitions. The control states are labeled with invariants (predicates over the continuous variables) and flow conditions (differential equations over the variables). Transitions may be labeled by both an event (from a finite set of events) and a jump condition consisting of both a guard predicate and assignments to the variables. The initial states are defined as combinations of control states and corresponding values for the variables. During execution, exactly one control state is active. While time evolves, the functions of the continuous variables are defined by the flow conditions of the active state. A state change may occur at any time, if the guard predicate of the transition evaluates to *true*, causing the variables to be assigned (potentially) new values causing a non-continuous change. The system may stay in a certain control state only as long as the invariant of this state holds.

Many interesting properties, such as reachability, of hybrid automata are undecidable [ACH+95]. Thus, subclasses of hybrid automata, such as rectangular automata and linear automata, have been

[1] There is another class of systems which combines discrete change with (limited) continuous behavior, called timed or real-time systems. In these systems time is modeled explicitly as a continuous value progressing at a uniform rate. As timed systems are easily embeddable into hybrid systems and do not contribute additional modeling concepts not already found in models of hybrid systems, we do not include them in the related work.

studied. They limit the general hybrid automata by allowing only a reduced language for the description of state invariants, flow conditions, and jump conditions. For these subclasses, interesting properties are often decidable (albeit being in an "expensive" complexity class), while their expressiveness, of course, is reduced.

The semantics of hybrid automata has been studied in great detail and the level of abstraction can be chosen by the modeler. Composition of multiple automata is performed by shared events, *i.e.*, transitions labeled with the same events are taken simultaneously. Access to the continuous state of other automata is not possible. Hybrid automata provide no notion of an interface. From the four criteria which are specific to our problem, the spatial and kinematic aspects can be expressed to some degree, however, the encoding of these properties in continuous variables is rather tedious and the lack of shared continuous variables prohibits composition in such a setup.

Semantics:	+
Abstraction:	+
Interfaces & Composition:	+
Spatial Relation:	o
Material & Material Flow:	-
Kinematics:	o
Communication & Energy:	o

Hybrid Input/Output Automata Nancy Lynch et al. propose hybrid input/output automata [LSV03] as a hybrid extension of input/output automata. These are described by a set of continuous variables (which are separated into three sets of input, output, and local variables), a set of actions (separated into input, output, and internal actions), a set of states (defined as a subset of valuations for the internal variables), a set of discrete transitions (defined as relation between state/action pairs and states), and trajectories which describe the allowed behavior of the variables. The initial states are just a subset of the state set. Contrary to the hybrid automata, the model contains no discrete control states. These are instead modeled as part of the continuous variables. A hybrid system is described by a set of hybrid I/O automata. Their composition is defined by a parallel execution where common actions are executed simultaneously, and the trajectories of all variables are chosen such that their projection to the variables of a single automaton is equal to the trajectories of this respective automaton. To make composition feasible, certain sets of variables and actions must be disjoint (for example the internal variables of one automaton may not overlap with the internal, input, or output variables of another automaton). Additionally, just as with plain I/O automata, the automata are required to be *input enabled*, *i.e.*, progress must be possible for any input action at any time. For the continuous part this also means that for the input variables all possible trajectories must be valid for the automaton.

The separation of input/output actions and variables provides a clearer notion of an interface compared to hybrid automata. Formal semantics, free choice in the level of abstraction, and composition are available as well. Compared to hybrid automata, the external actions and variables can be used to model both communication and transmission of energy. However, for the energy transmission, laws of preservation of energy have to be modeled explicitly.

Semantics:	+
Abstraction:	+
Interfaces & Composition:	o
Spatial Relation:	o
Material & Material Flow:	-
Kinematics:	o
Communication & Energy:	-

Hybrid Process Algebras There also is a couple of algebraic approaches for the description of hybrid systems. These include the work of Jan Bergstra and Kees Middelburg [BM05] and Pieter Cuijpers and Michel Reniers [CR05], which are both hybrid extensions of the process algebra ACP, and the model by Peter Höfner and Bernhard Möller [HM09], which is based on the concept of Kleene algebras. Hybrid behavior is captured by *trajectories*, which are piecewise continuous functions on intervals of a time domain, or by *flow clauses*, which are pairs of continuous variables and flow predicates (mostly differential equations) on these variables. Similar to process algebras, their hybrid counterparts are mostly used to understand the general nature of hybrid systems and are hard to directly apply to the actual modeling of real systems. They share the strengths of process algebras in terms of formal semantics, abstract modeling, and composition (while being weak at interface definition). In terms of the spatial/mechatronic requirements they can be compared to hybrid automata.

Semantics:	+
Abstraction:	+
Interfaces & Composition:	+
Spatial Relation:	o
Material & Material Flow:	-
Kinematics:	o
Communication & Energy:	o

HyCharts Thomas Stauner introduces HyCharts [Sta01], which consist of a continuous data-flow language (HyACharts) whose components are either described by another HyAChart or by a HySChart, which resembles a continuous extension of Statecharts. Interface definition and composition in HyACharts is similar to FOCUS using dense streams. Contrary to FOCUS, HyACharts apply a synchronous model of computation[2]. Causal loops have to be explicitly avoided by the modeler. HySCharts are hierarchic automata (similar to Statecharts but without AND states) that are augmented by so called *activities*, which are predicates associated to the states to describe the continuous evolution of variables associated with the component. The activities can be compared to the flow conditions found in hybrid automata. Stauner has a strong focus on a formal methodology for the development of hybrid systems, including a refinement calculus and property proofs. The strengths and weaknesses regarding our catalog of criteria are the same as with hybrid input/output automata, although HyCharts are more geared towards an actual engineering process.

Semantics:	o
Abstraction:	o
Interfaces & Composition:	+
Spatial Relation:	-
Material & Material Flow:	-
Kinematics:	-
Communication & Energy:	+

UML-RT & Bond Graphs UML-RT [SR98, Sel98] is an extension of a UML subset for modeling real-time systems based on ideas from ROOM[3]. *Capsules* are used for representing system components, their interfaces are described using *ports* which are connected via *connectors*. The behavior of a capsule is usually captured by a state machine. Bond graphs are used for graphical modeling of dynamic physical systems and are related to block diagrams. They were proposed by Henry Paynter [Pay60] and consist of blocks (representing system parts) connected by different types of *bonds*.

[2] For the description of hybrid systems there also is a variant of FOCUS on dense streams that is based on small delays in computation and thus is asynchronous [MS97].

[3] Real-Time Object-Oriented Modeling, see [SGW94]

The bonds are used to exchange power and are annotated with two physical quantities, called *effort* and *flow*.

Marcello Bonfé et al. discuss in [BFS05] an extension of UML-RT by using bond graphs. Therefore they introduce special ports called *power ports*, which are used to exchange power between capsules and are used in addition to *normal* ports which are still present for message exchange. The exchange of power via connectors is defined by *protocols*, which in the physical case are described using differential equations. The semantics of this integration leaves space for interpretation (partly due to the brevity of the paper). While the interface of the capsules can be described well, modeling on higher levels of abstraction is hindered by the requirement of explicitly specifying the power exchange very detailed. Support for spatial properties or material related issues is not provided, while capturing the energy flow is one of the strengths of their approach.

Mechatronic UML Mechatronic UML is a hybrid extension of UML that targets the description of mechatronic systems [GBSO04, BG05, BGH+07]. More precisely, only certain diagram types of the UML are used, such as component diagrams and state charts. The main goals are the integration of continuous control and dynamic reconfiguration with the existing discrete formalism. The approach provides formal semantics (by a transformation to hybrid automata), is not fixed to a single level of abstraction, and provides a notion of components and interfaces. All of the aspects related to space, including kinematics and material flow, are not considered in mechatronic UML.

Semantics:	+
Abstraction:	+
Interfaces & Composition:	+
Spatial Relation:	-
Material & Material Flow:	-
Kinematics:	-
Communication & Energy:	o

Summary Similar to models of pure software systems, models of hybrid systems usually do not directly support the special features required for space-intensive mechatronic systems. However, the encoding of those features often is simplified as at least some of the properties can be captured in continuous variables and the continuous motion of objects can often be easily expressed. The main problem of the hybrid approaches typically is the rather low level description language, which makes the formulation of certain properties (such as collision of arbitrary rotated objects) hard or even impossible. Spatial properties, when encoded in such a hybrid model, are usually not respected by the composition operators available. The notion of space allocation, collision, or complex transformations have to be mapped to the rather primitive (although expressive) means available in these models. Additionally, none of the techniques listed has an explicit notion of material, material flow, or kinematics. So, they can in principle be used to describe these machines, but place a high burden on the modeler who has to close the gap between these domain concepts and the provided (rather low level) modeling concepts.

3.3 Integrated Meta-Models

There are several approaches working on a common meta-model for mechatronic systems. Typically, these models are only defined on the syntactic level, while detailed semantics and especially the interplay between different parts of the meta-model are left unspecified.

MechaSTEP The goal of the MechaSTEP project [Dür99, Pav01] was the creation of a common data exchange format for construction and simulation tools used in the context of mechatronic systems. MechaSTEP builds upon the STEP standard (Standard for the Exchange of Product model data, ISO 10303), which is an exchange standard for CAD geometry data. STEP is extended by additional meta-model elements for electrics (*e.g.*, resistor or coil), control theory (*e.g.*, differential amplifier or addition), and hydraulics (*e.g.*, hydraulic cylinder). Interestingly, software is excluded and thus the behavior of a system can not be captured suitably. For the existing meta-model elements only physical laws are summarized, while the meaning of composition is left unspecified. Similarly, the level of abstraction is fixed to a very detailed level, where a resistor can be specified down to the coefficients of a temperature curve. Spatial and kinematic properties can be expressed, which is inherited from STEP, but this is limited to static geometry. Material generation and material flow is not considered in MechaSTEP.

Semantics:	-
Abstraction:	-
Interfaces & Composition:	-
Spatial Relation:	o
Material & Material Flow:	-
Kinematics:	o
Communication & Energy:	o

MEDEIA The *MEDEIA*[4] project [SRE+08] aims at providing a "formal framework for model-driven component-based development of embedded control" and is funded in the context of the EU's 7th Framework Programme. It's system model is based on so called *automation components* consisting of an interface described in terms of *ports* and *input assumptions* and *output guarantees*, and the *internal behavior* for which different specification techniques are suggested. These components can be organized hierarchically, where an automation component has access to its child components. All available documents only focus on the syntactic description of the model and do not give any hints on clearly defined semantics or details of the interface description. Similarly, the level of abstraction is fixed to the signal level and no higher-level signals are supported. The scope is limited to software and electronics without dealing with spatial properties, material, or energy transmission.

Semantics:	-
Abstraction:	o
Interfaces & Composition:	-
Spatial Relation:	-
Material & Material Flow:	-
Kinematics:	-
Communication & Energy:	-

[4] Model-Driven Embedded Systems Design Environment for the Industrial Automation Sector (`http://www.medeia.eu/`)

Semantics:	-
Abstraction:	-
Interfaces & Composition:	o
Spatial Relation:	+
Material & Material Flow:	-
Kinematics:	o
Communication & Energy:	o

AutomationML AutomationML[5] is an XML-based format for storing and exchanging construction and planning data in the factory automation domain [Dra10] developed by an industrial consortium. It integrates CAEX (Computer Aided Engineering Exchange) for the logical plant layout, COLLADA for 3D geometry and kinematics, and PLCopen for the description of program logics. The approach mostly describes the static structure of models in terms of XML elements used. Details on the semantics of the model elements is not provided, only the PLCopen part is based on programming languages which are standardized in IEC 61131-3 [IEC03]. These languages, however, do not provide a formal underpinning and limit the logics description and thus the entire AutomationML model to a single (fairly detailed) level of abstraction. Spatial and kinematic properties are covered by COLLADA, but material an material flow are not covered by AutomationML and thus also dynamic kinematics for material objects are missing. The language supports a simple interface definition based on ports.

Summary The approaches summarized in this section attempt to integrate the discipline specific models into a single one by providing a common data format or meta-model. While this can be seen as a first step towards integrated models, the interplay of the model elements (or more formally the model's semantics) are left undefined, although this is the more relevant ingredient to ensure consistent use across disciplines. Additionally, while some of the models deal with spatial aspects (mostly based on CAD geometry) they provide no deep integration with the behavior model or support for material flow.

3.4 Virtual Commissioning

Virtual commissioning is the process of testing a machine's controller software in conjunction with a simulated model of the machine (including simulated sensors and actors). The controller either runs on the real controller hardware (then referred to as hardware-in-the-loop testing) or a PLC emulator (software-in-the-loop test). The machine model has to support not only the structure of the machine's hardware, but especially its behavior to be applicable for virtual commissioning. Thus, and also because some of the approaches in this area allow for specification of (parts of) the controller behavior, these models are a suitable source when looking for integrated behavior models.

[5] http://www.automationml.org/

Semantics:	-
Abstraction:	o
Interfaces & Composition:	o
Spatial Relation:	o
Material & Material Flow:	o
Kinematics:	o
Communication & Energy:	o

SEMI The SEMI project (Simultaneous Engineering for Development of Machines with Micro-systems) [ABH+99] is one of the earlier works in the area of virtual commissioning. Although a significant part of the project focused on the coupling of the controller hardware and the machine model using a field bus, the project also defined a tool supported modeling technique for describing the machine's behavior. The model was described using ROOM [SGW94] for the logical structuring of the model, and C++ for the actual behavior. These models could then be used to generate C++ programs, which could be compiled to yield the executable machine simulator.

Due to the nature of ROOM and C++, support for continuous changes within the machine, material flow, or collision detection and response are not directly supported by the model but have to be *programmed* by the modeler. As the model is built on top of C++, it inherits its semantics, which are, while being standardized [ISO03b], not captured formally but rather defined in terms of the compiler's output. Energy flow can be expressed using ROOM's bindings (connections) and kinematics are at least supported for displaying purposes, where a 3D model of the machine is animated based on the model's output.

Semantics:	-
Abstraction:	o
Interfaces & Composition:	o
Spatial Relation:	o
Material & Material Flow:	o
Kinematics:	+
Communication & Energy:	o

Virtual Commissioning Tools Today the area of virtual commissioning is heavily influenced by commercial tools, which here are discussed together. The four most commonly used tools are briefly summarized in alphabetical order. **Process Simulate Commissioning**[6] by *Siemens PLM* (formerly Technomatix) describes the machine model based on a 3D geometric model annotated with constraints (joints with degrees of freedom). Behavior is specified in terms of a sequence diagram which only allows to capture a single flow of motions (no branching based on conditions). **SIMIT**[7][8] by *Siemens AG* uses data-flow graphs to express the machine's behavior. Nodes in the graph are either predefined blocks (mostly simple arithmetic and logic expressions) or can be programmed in a C-like procedural language. **Virtuos**[9] by *ISG* (originally developed at the ISW[10] of Universität Stuttgart) specifies the machine's behavior using data-flow graphs complemented by state machines. In addition, it couples a 3D geometry model (for visualization only) and a very simple material flow model (cuboids and collision detection for motion in the direction of the coordinate axes only). **WinMOD**[11] by *Mewes & Partner GmbH* is similar to SIMIT, but provides a

[6] http://www.plm.automation.siemens.com/en_us/products/tecnomatix/robotics_automation/commissioning.shtml

[7] Earlier versions were also available under the name *Sinumerik Machine Simulator*

[8] http://www.industrysolutions.siemens.com/industrial-services/cross-industry/de/automation_it/simulation_test/simit.htm

[9] http://www.isg-stuttgart.de/virtuos.html

[10] Institut für Steuerungstechnik der Werkzeugmaschinen und Fertigungseinrichtungen

[11] http://www.winmod.com/

larger collection of primitive data-flow nodes (which is required as no programming language is included).

As could be expected from tools used in the industry, all of them allow to model a machine's behavior detailed enough for simulation purposes. Most of them are based on a variant of data-flow networks and also allow the integration with 3D geometry models for visualization of the systems spatial state. However, as found during an unpublished case study by the *iwb*[12] and the author, these tools do not support feedback from the geometry model to the simulation and also the material flow models (if any) are very limited. For typical production machines the effort for modeling congestion on belt conveyors or the activation of proximity sensors can be as much as 80% of the overall modeling effort. Modeling at higher levels of abstraction is not supported well, as modeling has to happen at the controller's signal level. Finally, the semantics are only defined in terms of the tool implementation (*i.e.*, the model's execution) and broad intuition given in the manuals.

AQUIMO The BMBF[13]-funded project FÖDERAL[14] developed a methodology for building models of mechatronic systems from so called *mechatronic components*. These components consist of a set of partial models, *e.g.*, for PLC code and electric connection diagrams. A tool then allows to generate the full models (PLC code or connection diagram) from a set of connected components. Therefore the partial models are copied to a single model and connected according to rules that are specific to the type of models used and hard-wired into the tool.

Semantics:	-
Abstraction:	o
Interfaces & Composition:	o
Spatial Relation:	o
Material & Material Flow:	-
Kinematics:	o
Communication & Energy:	o

The successor project AQUIMO[15] (also BMBF-funded) applies the ideas from FÖDERAL to virtual commissioning models[16] described using *Virtuos* (which was described in the previous section). The goal is to describe a mechatronic system by individual components which are specified with Virtuos. The components can be inserted and connected in a 3D geometric view of the model. This model allows the generation of a complete Virtuos model, which then can be simulated or used for virtual commissioning. As Virtuos is not prepared for this kind of components, many details have to be annotated to the components. For example, a simple component can only be used in a specific orientation. If other orientations shall be available, the required modifications to the underlying (partial) Virtuos model have to be described in the component as well.

Overall, the AQUIMO approach inherits some of the problems of Virtuos, such as the absence of formal semantics. Some issues, such as abstraction and component interfaces are approached by AQUIMO, but their solution introduces some new problems. The notion of a component used even

[12] Institute for Machine Tools and Industrial Management of Technische Universität München
[13] Bundesministeriums für Bildung und Forschung, Federal Ministry of Education and Research
[14] `http://www.foederal.org/`
[15] `http://www.aquimo.org/`
[16] As there are no publications on the project, the description given here is based only on a common workshop with the AQUIMO project's participants on 3rd November 2009.

reduces the functional range of Virtuous, as for example no compositional notion of material flow is supported, thus eliminating the (already limited) material flow support of Virtuous.

Summary Overall, approaches used for virtual commissioning target a similar problem as the one of this thesis. While being practically applicable, they show several shortcomings when used in practice. The two most obvious ones are the weak integration of the system's geometry, which is usually only used for visualization, and the lack of a compositional and expressive material simulation system. The lack of formal underpinning may not be an obstacle for the application in virtual commissioning, but hampers the use of these models for different applications.

3.5 Simulation-Centric Approaches

While the approaches discussed in the previous sections aim at the simulation of the machine's hardware only, this section summarizes models which are primarily used for a simulation of the entire system (including controller software). Thus, all of the models listed next are *executable* in some sense.

Quantitative Material Flow Simulation Peter Struss et al. describe in [SKSV08] a quantitative material flow model for bottling plants. In these bottling plants more than 100,000 bottles can be processed each hour. Their model abstracts the plant by transportation elements (which also act as buffers as they can contain a large number of bottles) and connectors between them. Each transportation element is described by several parameters, such as the *potential inflow* and *outflow* (in bottles per second), or the current and maximal number of stored bottles. As bottles are not modeled individually but rather treated as a flow, small-scale interaction with material can not be captured. Still, their model can simulate effects such as gaps in the material stream or tailbacks. A numerical simulation of the material flow and the plant's throughput can be performed using Matlab/Simulink[17].

Semantics:	+
Abstraction:	o
Interfaces & Composition:	+
Spatial Relation:	-
Material & Material Flow:	o
Kinematics:	-
Communication & Energy:	o

The assessment of this model within our criteria catalog is not easy, as the model serves a slightly different purpose than many of the other techniques discussed here. Especially, due to its very high level of abstraction, some aspects are trivial, while others are not relevant. For example, both the semantics of the model and the notion of interfaces is clearly defined (and simple compared to other models), while the limitation to the single (very high) level of abstraction may be a drawback during systems development. Similarly, spatial aspects are not contained (but of course this has been one goal of the authors), which also makes kinematics and more fine-grained material interaction impossible to express.

[17] http://www.mathworks.com/products/simulink/

Semantics:	+
Abstraction:	o
Interfaces & Composition:	+
Spatial Relation:	o
Material & Material Flow:	-
Kinematics:	-
Communication & Energy:	+

Modelica The language *Modelica* [Til01] was created in 1997 by a consortium called the *Modelica Association*[18]. It supports modeling of complex physical systems with continuous and discrete parts. Models can be split into (potentially hierarchical) components which provide typed *connectors*. A component is described using a set of differential equations describing the flow between the connectors and local (state) variables. A large library of predefined components as well as commercial modeling and simulation tools are available. For collision detection and response an extension is proposed in the PhD thesis of Vadim Engelson [Eng00].

The disadvantage of Modelica is the high level of detail needed for the system description: masses, torques of inertia, and friction coefficients are not known in the early development phases. Furthermore this makes it hard to model parts of the machine at different levels of abstraction, as we always have to respect physics (which we might want to violate in early models). While Modelica and Engelson's extension support geometric objects and (physically correct) collision response, a simple collision detection without reaction (for example to model a light barrier) is not supported. Similarly, the support is limited to single unconnected objects, thus neither kinematic relationships nor material flow scenarios can be easily expressed.

Semantics:	-
Abstraction:	o
Interfaces & Composition:	o
Spatial Relation:	+
Material & Material Flow:	o
Kinematics:	+
Communication & Energy:	-

Physics Simulation in Movies and Games Driven by large amounts of money spent on the production of movies and computer games, there is lots of work on physical simulation for achieving special effects in both of them. Solutions in this area range from general physics simulations, such as the Havok engine[19] which is commonly used for video games and movies, to more specialized simulation systems, such as MASSIVE[20], as system for crowd simulation which is most famous for its application in battle scenes in the 2001 movie adaption of Tolkien's *Lord of the Rings* [Tol54]. The main difference to physical simulation as found in car crash simulations or finite element method (FEM) simulations is that these engines are not limited to pure mechanics/physics, but also deal with logical interaction. In game engines this is the interaction with the player and other actors controlled by the computer, while for movies behavior modeling is used to unburden the modeler from describing too many details (for example when animating large crowds).

While it is hard to summarize the entire field of interactive physics simulation, most of these simulation engines share the property that the exact semantics are not formally defined but rather given by the simulation program. Logics and interactions are typically described by proprietary data-flow or scripting languages. As the goal of the model is to produce realistic looking visual output, the abstraction level is typically chosen to match exactly this goal, which is not necessarily the level that would be chosen for systems development. Similarly, a notion of components and interfaces is

[18] `http://www.modelica.org/`
[19] `http://www.havok.com/`
[20] `http://www.massivesoftware.com/`

typically present in a simple form to allow reuse of partial models, but not in a rigorous manner as expected for systems engineering. These engines are very detailed in representing spatial aspects and kinematics, and most of them allow dynamic generation of objects (particle systems) which can capture at least certain aspects of the material flow. Explicit communication and exchange of energy is neglected.

Summary The common property of the models described here is that their primary intent is the simulation of the models (and not formal analysis). As the scope of simulation systems is very wide, there are solutions for many very specific modeling and simulation problems. Still, an approach combining spatial/kinematic properties and material flow with rigorous semantics and interfaces is missing.

3.6 Mechatronic and Systems Engineering Models

Many models aim at the systems engineering process, *i.e.*, are meant to be used during the development of multi-disciplinary models. This section picks those approaches that are intended to be used for mechatronic systems.

SysML SysML (Systems Modeling Language) [Obj08] is a language for modeling systems proposed by the Object Management Group. Development of SysML started in 2001 and was coordinated by the *OMG Systems Engineering Domain Special Interest Group*. In 2006 the first version was finalized and in 2008 the current version 1.1 was published. SysML is based on a subset of UML 2.0 which is extended with two additional diagram types and several stereotypes. For example there are additional port types, called *flow ports*, which are used to describe exchange of energy or matter. The high number of new elements in addition to the existing ones from UML lead to the common criticism of *language bloat*, *i.e.*, increased complexity in learning and applying SysML due to its extent[21].

Semantics:	o
Abstraction:	+
Interfaces & Composition:	+
Spatial Relation:	-
Material & Material Flow:	o
Kinematics:	-
Communication & Energy:	+

As with UML, the semantics of SysML is textually described in the specification, but is far from being formal and complete. The level of abstraction of the model can be chosen freely within certain limits and system blocks can specify their interfaces in terms of ports. While the exchange of energy and matter can be modeled using flow ports, the material flow is only a logical one and spatial properties and kinematics are not supported at all.

[21] This is for example mentioned as a problem in the FAQ of the *SysML Forum* (`http://www.sysmlforum.com/FAQ.htm`), which belongs to the *SysML - Open Source Specification Project* (`http://www.sysml.org/`) – an organization consisting of serveral industrial partners that contributed to the SysML specification.

Semantics:	-
Abstraction:	o
Interfaces & Composition:	-
Spatial Relation:	-
Material & Material Flow:	-
Kinematics:	-
Communication & Energy:	o

Lippold Christian Lippold proposes an interdisciplinary approach for the development of mechatronic systems in [Lip00]. The general idea is to combine hierarchical decomposition and description techniques from different domains, including block diagrams, class diagrams, bond graphs, and state charts. Unfortunately, the thesis only provides a high level view of his approach and does not explain how these different techniques should be combined. Consequently, the semantics of the underlying behavior model is undefined. Although different views on a system are defined (function, action, behavior), the role of abstraction is left sketchy and a clear notion of interfaces is not established. The aspects of space, geometry, kinematics, and material flow are not considered.

Semantics:	-
Abstraction:	o
Interfaces & Composition:	-
Spatial Relation:	o
Material & Material Flow:	o
Kinematics:	o
Communication & Energy:	o

Kallmeyer Ferdinand Kallmeyer describes an approach for modeling *feasible solutions*[22] in [Kal98]. It is based on a functional decomposition, where the individual functions can later be described by various *aspects*. For the behavior description both state charts and Petri nets are suggested. The functions can also be augmented by aspects for kinematic and dynamic simulation. The interaction between the different aspects of a function and between different functions is only described by box and arrow diagrams with unclear semantics (although the arrows are labeled with suggestive names). Abstraction can be influenced by the degree of functional decomposition. A formal notion of interfaces is not provided. While the submodels for the functions' aspects can include spatial properties and details on the kinematics, the interplay with the behavior model remains unexplained. For the flow of energy and material also box and arrow diagrams are used, which depict the logical flow, but are not integrated with the other aspects of a function.

Semantics:	-
Abstraction:	+
Interfaces & Composition:	+
Spatial Relation:	-
Material & Material Flow:	o
Kinematics:	-
Communication & Energy:	o

Gehrke Matthias Gehrke proposes an approach for describing mechatronic systems based on function hierarchies and system structures [Geh05]. The individual functions in a function hierarchy are described by a verb and subjects for the function's input and output. Both the verb and the subjects are chosen from a predefined (but possibly project specific) vocabulary. The system structure is given by flow diagrams which depict the exchange of energy, matter, and information between system elements. While for all the models involved formal meta-models are given, the exact semantics is omitted. This is not problematic for his approach, as the goal there is an algorithm that supports the systematic transition from function hierarchies to system structures. As long as the syntactic elements are used consistently, *i.e.*, have the same semantics, the algorithm will work.

For our goal of describing the behavior of a system, the lack of semantics is problematic. Different levels of abstraction are supported and the interface of system elements can be described via ports.

[22] German: *Prinziplösungen*

The ports differentiate between input and output ports and the type exchanged (*i.e.*, energy, matter, or information), which can be further refined. Gehrke's approach ignores the topics of space and kinematics, while material and energy flow is only dealt with on a very basic (logical) level.

Summary The approaches presented in this section are influenced by systems engineering. What they have in common is a lack of semantics and often also limitations in terms of interface description. Interestingly, the properties relevant for automation systems, *i.e.*, spatial relationships and a detailed material flow, are not supported at all by these approaches.

3.7 Summary

In this chapter we presented existing approaches for the description of software and mechatronic systems. As each of the individual sections have their own summary, here we concentrate on the big picture. The assessments with respect to our criteria from Section 2.4 are summarized in Table 3.1 to ease comparison. What is clearly visible from this table is that most approaches are weak when it comes to spatial properties and material flow, which are both usually not supported or only supported on a very abstract logical level that is not sufficient for capturing certain interactions. Those techniques that support spatial aspects, material, and their integration with the behavior model, in contrast, lack formal semantics and are often weak in terms of interface definition and abstraction. The modeling theory presented in the next chapters is meant to fill this gap, *i.e.*, provide an approach that is strong in terms of semantics, abstraction, and composition/interfaces, but also integrates the spatial aspects.

	Semantics	Abstraction	Interfaces & Composition	Spatial Relation	Material & Material Flow	Kinematics	Communic. & Energy
Formal Models of Software							
Statecharts	+	+	-	-	-	-	-
Input/Output Automata	+	+	+	-	-	-	-
Petri Nets	+	+	o	o	o	-	o
Process Algebras	+	+	o	-	-	-	-
Lustre	+	+	+	-	-	-	o
FOCUS	+	+	+	-	-	-	o
Models for Hybrid Systems							
Hybrid Automata	+	+	-	o	-	o	-
Hybrid Input/Output Automata	+	+	+	o	-	o	o
Hybrid Process Algebras	+	+	o	o	-	o	-
HyCharts	+	+	+	o	-	o	o
UML-RT & Bond Graphs	o	o	+	-	-	-	+
Mechatronic UML	+	+	+	-	-	-	o
Integrated Meta-Models							
MechaSTEP	-	-	-	o	-	o	o
MEDEIA	-	o	-	-	-	-	-
AutomationML	-	-	o	+	-	o	o
Virtual Commissioning							
SEMI	-	o	o	o	o	o	o
Virtual Commissioning Tools	-	o	o	o	o	+	o
AQUIMO	-	o	o	o	-	o	o
Simulation-Centric Approaches							
Quantitative Material Flow Simulation	+	o	+	-	o	-	o
Modelica	+	o	+	o	-	-	+
Physics Simulation in Movies and Games	-	o	o	+	o	+	-
Mechatronic and Systems Engineering Models							
SysML	o	+	+	-	o	-	+
Lippold	-	o	-	-	-	-	o
Kallmeyer	-	o	-	o	o	o	o
Gehrke	-	+	+	-	o	-	o

Table 3.1: Summary of the assessment of related work

4 Space and Time

When modeling spatio-temporal systems, *i.e.*, systems in space over time, the selection of formal models for space and time is crucial, as it has a major impact on *what* can be modeled on *which level of detail (or abstraction)* and how complicated it is to express certain properties. Furthermore, it affects the modeling theory by making certain proofs more or less hard or even disabling certain desired properties of the theory.

This chapter provides a short overview on existing and widely used models of both space and time. It describes the space/time model chosen for our model of spatio-temporal systems, which is then introduced in the following chapter. The main contribution of this chapter is the notion of a transformable collision space (Section 4.2.2), a structure of space which is simple, yet sufficiently expressive for our purpose. To better understand the ideas behind the transformable collision space and also to form the foundation for later chapters, several properties of the transformable collision space are discussed.

4.1 Preliminaries

Before discussing space and time, we recapitulate some mathematical definitions and notations used in this and the following chapters, to ensure familiarity with them. We denote the natural[1], rational, and real numbers by $\mathbb{N}$, $\mathbb{Q}$, and $\mathbb{R}$, respectively. The set of Boolean values is denoted by $\mathbb{B} := \{\boldsymbol{t}, \boldsymbol{f}\}$, where $\boldsymbol{t}$ is the abbreviation for $x \vee \neg x$ and $\boldsymbol{f} := \neg\boldsymbol{t}$. For a set S we denote by $\mathcal{P}(S)$ its *power set* and by $\mathcal{P}_{\text{fin}}(S) = \{T \subseteq S \mid |T| \in \mathbb{N}\}$ the set of all *finite* subsets of S.

For a partial order $\leq$ on S and a subset $T \subseteq S$ we denote by $\inf(T)$ its *infimum*, *i.e.*, the largest element in S (not necessarily in T) which is a lower bound for all elements in T, and by $\sup(T)$ its *supremum*, *i.e.*, the smallest element in S which is an upper bound for all elements in T, if these exist and are unique. These are sometimes also called *greatest lower bound* (glb) and *least upper bound* (lub).

For a function $f : D \to V$ we denote its domain D by $\text{dom}(f)$ and its image by $f(D)$, *i.e.*, we extend function application to sets element-wise. Instead of $f(x)$ we sometimes write $f.x$ to reduce

[1] There sometimes is disagreement on whether 0 is a natural number or not, and there are good reasons for both cases. Here we use the convention that $0 \in \mathbb{N}$.

the number of parentheses and improve readability. For $\tilde{D} \subseteq D$ we denote by $f|_{\tilde{D}}$ the limitation of f to $\tilde{D}$ defined by

$$f|_{\tilde{D}} : \tilde{D} \to V \ ; \qquad f|_{\tilde{D}} : d \mapsto f(d) \ .$$

For two functions $f : T \to U$ and $g : S \to T$ we denote their composition[2] by $f \circ g$, defined by

$$f \circ g : S \to U \ ; \qquad f \circ g : s \mapsto f(g(d)) \ .$$

The *inverse* function of f (if it exists) is denoted by f^{-1} and uniquely defined by the equation $\forall x \in \mathrm{dom}(f) : \ (f^{-1} \circ f)(x) = x$. For a square matrix $M \in \mathbb{R}^{n \times n}$ we denote its *inverse* (if it exists) by M^{-1}, the *transposed* matrix by M^T, and its *determinant* by $\det(M)$.

4.2 Formalizing Space

If we want to capture spatial properties of systems in a formal model, we need a formal model of space to build upon. While such a model of space has to be detailed enough to describe the properties of interest, our goal is to keep it as minimal and general as possible. This avoids the occlusion of our theory (*c.f.*, Chapter 5) by superfluous concepts and notations, allows the application to different settings, and also gives insights in how far we can simplify space. The latter is important for proving certain properties or implementing automatic decision procedures for them.

4.2.1 Common Notions of Space

This section presents an overview of space models used in different areas of both research an practice. The models are discussed as they serve as a starting point for our model and allow to better explain the differences to our model later on.

Set Theory Set theory, as initiated by Cantor and Dedekind, is often called the foundation of modern mathematics. Although there are different axiom systems, the one commonly used is the Zermelo-Fraenkel system with the axiom of choice (ZFC). This leads to the *well-known* operations on sets: intersection, union, complement, set difference, Cartesian product, and power set.

Based on systems of sets, mathematical topology defines several mathematical spaces, the most basic one being the *topological space* which defines for a universe of points a family of open sets closed under intersection. While topological space allows the definition of connectedness and continuousness, it can not be used to *measure*. For this, the *metric space* is defined by a universe of points and a *metric* (or distance function), which is a symmetric mapping of pairs of points into the non-negative reals that respects the triangle inequality and is 0 only for equal points. The canonical examples of metric spaces are the discrete metric space, with an arbitrary point set and a metric

[2] Some textbooks use the reversed definition of the composition operator, *i.e.*, the left function would be applied first.

which returns 0 for equal and 1 for inequal points, and the Euclidean space. Each metric space also induces a topological space.

Linear Algebra and Analytic Geometry The mathematical branch of linear algebra studies systems of linear equations and vector spaces. A *vector space* $(V, +)$ over a field F is an Abelian group with a binary operation $F \times V \to V$ (scalar multiplication) which follows axioms of associativity and distributivity and preserves the neutrality of 1_F. A *normed vector space* is a pair of a vector space V and a norm, *i.e.*, a function from V into the non-negative reals, such that it is 0 only for the vector 0_V and respects positive scalability and the triangle equation. These spaces are interesting as by the norm a metric is implied and thus each normed vector space is a metric space as well. The canonical example for a normed vector space is again the n-dimensional Euclidean space.

The Euclidean space is a special case of n-dimensional coordinate spaces over a field F, denoted by F^n, which occur in the study of systems of linear equations. In this space, each equation in n variables describes a $(n-1)$-dimensional hyper-plane, which itself is a (sub) vector space. The solutions of a system of equations is then described by the intersection of these hyper-planes. While a linear equation describes a hyper-plane, a linear inequality describes an open or closed half-space, limited by the hyper-plane of the corresponding equation. The intersection of half-spaces defined by a system of linear inequalities results in a *convex polyhedron* which can be unbounded or bounded (in which case it is called a *polytope*). These structures are studied in the field of *linear programming* an can be used as a linear approximation of convex subsets in coordinate spaces.

The concepts of linear equations and inequalities for the description of subsets of space can be extended to higher-order polynomials. The resulting sets are a subject in the area of *analytic geometry*. Practical applications are the higher order surfaces and splines found in computational geometry, which are discussed later on.

Spatial Logics The goal of spatial logics is to capture and reason about spatial properties in a formal logical framework. They are usually built upon topological or metric spaces and define or limit the operators that are allowed to formulate predicates. These logics are built upon an underlying spatial model, which is either *point-based*, *i.e.*, reasoning is performed on sets of *points*, or *region-based*, *i.e.*, the underlying set consists of *regions of space* and reasoning is on single elements from this set. Currently most research deals with point-based logics over topological or metric spaces [KWS$^+$03, WZ05, APHvB07]. Typical applications of spatial logics are mostly in the area of artificial intelligence to support automatic reasoning about space [EG97].

Examples of spatial logics include $\mathcal{S}4_u$, which besides the *usual* Boolean connectives (which in this context are often rather written as set complement, intersection, and union) provide terms that express the interior and the exterior of a set, and the subset relation. These terms are sufficient to also express existential and universal quantification. Another spatial logic, called $\mathcal{RCC}-8$, is based on eight binary predicates (such as checking for disconnectedness, external connectedness, etc.; see

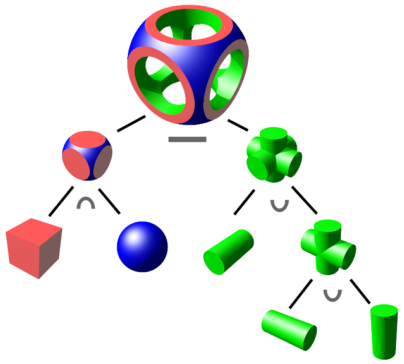


Figure 4.1: Sketch of the construction of a CSG model by set operations
Image source: `http://commons.wikimedia.org/wiki/File:Csg_tree.png`

[KKWZ07] for the full list) and can be embedded into $\mathcal{S}4_u$ [Ben94]. There also is ongoing work in the combination of temporal and spatial logics with the goal to express spatio-temporal properties, but current work indicates that it is hard to find a combination which is sufficiently expressive while still being decidable [GKK+05, KKWZ07].

Computer-Aided Geometric Design In mechanical engineering, space is usually described using tools that are summarized as computer-aided geometric design (CAD[3]), which can be seen as the descendants of technical drawings. The models described by these tools serve as documentation for construction and assembly, and as input for other analysis techniques, such as multi body simulation (MBS) or finite element method (FEM).

Although two dimensional descriptions still exist, most models today are described with respect to the three dimensional Euclidean space. CAD models are described as a collection of *parts*, which are shapes of rigid objects, and *constraints* between them, which restrict and often uniquely define the relative positions of the parts. Examples of constraints include alignment of faces, edges, or corners, or more generally relations between the local coordinate systems of parts. Often, additional information which is not directly space related is included as well, such as the material used, the mass of a part, or the part number of a supplier for standard parts.

For the description of shapes there are two general representations which can be converted to each other (although there are often practical limitations in the tools). One approach is the direct description of the volume of a part in terms of parametric primitives (boxes, cones, etc.) and operations on volumes, such as set intersection, union, or difference. This volume based description is referred to as *constructive solid geometry (CSG)* [Hof93] (*c.f.*, Figure 4.1). The complementary approach,

[3] Usually the word *geometric* is dropped and thus the term is abbreviated as CAD, but there also is CAD software for electrical or fluid design, which often has more of a logic and connection oriented view.

called *boundary representation (BREP)*, describes the surface of a part, that is the boundary between the inside and the outside. These surfaces can be delimited by polygonal faces, or by using spline based parametric two dimensional curves (a commonly used construct in this category is the *nonuniform rational B-spline (NURBS)*), which are part of multiple standards in the field of CAD. While CSG simplifies many volume related calculations, BREPs are often easier to visualize and simplify the support of complex modeling operations, such as extrusion, chamfering, or blending.

4.2.2 Transformable Collision Space

All of the approaches summarized in the previous section can be used to capture certain spatial properties. In the next paragraphs we describe the core structure we are using for reasoning about space, which also is one of the foundations of the model described in Chapter 5. The goal is to find a minimal structure supporting a description of the following properties:

- Provide a notion of parts in space (*spatial objects*),
- support assembly of parts from smaller parts (*spatial composability*),
- describe the transformation of parts (*spatial motion*), and
- check for the overlapping of parts (*spatial collisions*).

We capture our solution in the following definition:

Definition 4.1 [Transformable Collision Space]
A tuple $(V, 0_V, \bowtie, \sqcup, \mathcal{T})$ with

- a non-empty set of *volumes* V,
- the *empty volume* $0_V \in V$,
- the *collision relation* $\bowtie \subseteq V \times V$,
- the *volume union operator* $\sqcup : V \times V \to V$,
- and a set of *volume transformations* $\mathcal{T} \subseteq \{V \to V\}$ containing the identity function

is called a *transformable collision space*, iff the following conditions are met:

1. $\bowtie$ is symmetric, reflexive for non-empty volumes, and the empty volume collides with nothing, *i.e.*, for all $u, v \in V$

$$u \bowtie v \;\Leftrightarrow\; v \bowtie u \;,$$

$$v \neq 0_V \;\Leftrightarrow\; v \bowtie v \;, \tag{4.1}$$

$$\neg(0_V \bowtie v \;\vee\; v \bowtie 0_V) \;. \tag{4.2}$$

2. $(V, \sqcup)$ is a commutative monoid with identity element 0_V and $\sqcup$ is idempotent.

3. The following law of distributivity holds for all $u, v, w \in V$:

$$u \bowtie v \ \vee \ u \bowtie w \quad \Leftrightarrow \quad u \bowtie (v \sqcup w) \tag{4.3}$$

4. The transformation functions with function composition $(\mathcal{T}, \circ)$ form a group, *i.e.*, $\mathcal{T}$ contains the identity function and the inverse transformation for each $t \in \mathcal{T}$.

5. Transformations do not distort volumes, *i.e.*, for all $T \in \mathcal{T}$ and $u, v \in V$

$$\begin{aligned} u \bowtie v \quad &\Leftrightarrow \quad T(u) \bowtie T(v) \ , \\ T(u \sqcup v) \quad &= \quad T(u) \sqcup T(v) \ . \end{aligned} \tag{4.4}$$

⌟

The set of volumes colliding with a given volume v is denoted by $\mathcal{C}_{\bowtie}(v)$:

$$\mathcal{C}_{\bowtie}(v) \quad := \quad \{u \in V \mid u \bowtie v\}$$

Equation 4.3 guarantees that $\mathcal{C}_{\bowtie}(u \sqcup v) = \mathcal{C}_{\bowtie}(u) \cup \mathcal{C}_{\bowtie}(v)$, *i.e.*, the collision set for assembled volumes can be calculated piece-wise.

The reason to build our model of space upon *collision* instead of *intersection* is that collision is the more primitive of them. Intuitively both are related, as we expect two volumes to collide if their intersection is non-empty. However, collision is only a decision problem, while intersection requires the computation of a new volume. Additionally, our definition of space does not require the volumes to be closed under intersection, which can be useful for very abstract and coarse models of space. As shown in Chapter 5, collision is sufficient for describing the behavior of space-intensive systems. Furthermore, the limitation to collision simplifies an implementation of the theory as typical libraries for computational geometry support both union and collision, but intersection only in rare cases.

From the definition of a transformable collision space we can derive some common properties.

Corollary 4.2
$v \bowtie (v \sqcup u)$ *holds for* $v \neq 0_V$, i.e., *merging parts does not make them smaller (in terms of collidability).* ⌟

Proof. Equations 4.1 and 4.3. □

Corollary 4.3
All transformations $T \in \mathcal{T}$ *are idempotent for* 0_V, i.e., $T(0_v) = 0_v$. ⌟

Proof. Equations 4.2 and 4.4. □

Spatial Selection When describing properties of spatial models, often the selection of a certain set of spatial objects or volumes is required. These selections can be used to limit the scope of properties or characterize sets of volumes.

Definition 4.4
For a fixed transformable collision space $(V, 0_V, \bowtie, \sqcup, \mathcal{T})$ a predicate $\sigma : V \to \mathbb{B}$ is called a *spatial selection predicate*. The set of all possible predicates is denoted by $\mathcal{S}(V)$. We denote the *characteristic set* of a spatial selection predicate σ by $\chi(\sigma)$, which is defined as

$$\chi(\sigma) := \{v \in V \mid \sigma(v)\} \ .$$

⌟

Remark 4.5
We extend a spatial selection predicate $\sigma : V \to \mathbb{B}$ to a predicate on sets of volumes $\sigma' : \mathcal{P}(V) \to \mathbb{B}$ by $\sigma' : \{v_1, \ldots, v_n\} \mapsto \sigma(v_1 \sqcup \ldots \sqcup v_n)$, with the special case of $\sigma' : \emptyset \mapsto \sigma(0_V)$. Usually we just write σ instead of σ'. ⌟

One way to define a spatial selection predicate is by the definition of other volumes, which *select* all volumes they touch (*i.e.*, collide with). This is formalized in the following definition:

Definition 4.6
For a fixed transformable collision space $(V, 0_V, \bowtie, \sqcup, \mathcal{T})$ and a finite number of *selection volumes* $s_1, \ldots, s_n \in V$ the *collision selection predicate* $\sigma_{\bowtie}(s_1, \ldots, s_n) : V \to \mathbb{B}$ is the spatial selection predicate defined by

$$\sigma_{\bowtie}(S_1, \ldots, S_n)(v) \quad :\Leftrightarrow \quad \bigwedge_{i=1}^{n} S_i \bowtie v \ ,$$

i.e., volumes colliding with all of $s_1, \ldots, s_n$ are selected. ⌟

Approximation Using the basic axioms we can explain the meaning of spatial approximation in our framework.

Definition 4.7 [Over-Approximation Relation]
For a transformable collision space $(V, 0_V, \bowtie, \sqcup, \mathcal{T})$ the *over-approximation relation* $\sqsubseteq \ \subseteq V \times V$ is defined by

$$u \sqsubseteq v \quad :\Leftrightarrow \quad \forall w \in V : \ w \bowtie u \implies w \bowtie v \ .$$

⌟

So v over-approximates u, if each object colliding with u also collides with v. If $u \sqsubseteq v$ we call v an *over-approximation* of u and u an *under-approximation* of v.

Remark 4.8
An equivalent formulation for over-approximation can be achieved using the collision sets:

$$u \sqsubseteq v \qquad \Leftrightarrow \qquad \mathcal{C}_{\bowtie}(u) \subseteq \mathcal{C}_{\bowtie}(v)$$

⌟

Corollary 4.9
The over-approximation relation $\sqsubseteq$ is a preorder on V. ⌟

Proof. Reflexivity and transitivity are easily seen from the collision set formulation. □

Definition 4.10 [Indistinguishable Volumes]
Let $(V, 0_V, \bowtie, \sqcup, \mathcal{T})$ be a transformable collision space and $u, v \in V$. The volumes u and v are called *indistinguishable*, iff $u \neq v$ and $\mathcal{C}_{\bowtie}(u) = \mathcal{C}_{\bowtie}(v)$. ⌟

Example 4.11 [Indistinguishable Volumes]
Let $V = \mathcal{P}(\{0, 1\})$, $\bowtie = (V \setminus \{\emptyset\}) \times (V \setminus \{\emptyset\})$, and $\mathcal{T}$ the set containing the identity function. Then $(V, \emptyset, \bowtie, \cup, \mathcal{T})$ is a transformable collision space (where $\cup$ is the usual set union) and $\{0\}$ and $\{1\}$ are indistinguishable. ⌟

Lemma 4.12
Let $(V, 0_V, \bowtie, \sqcup, \mathcal{T})$ be a transformable collision space without indistinguishable volumes. Then $\sqsubseteq$ is a partial order on V. ⌟

Proof. Following Corollary 4.9 we only have to show antisymmetry, so let $u, v \in V$ with $u \sqsubseteq v$ and $v \sqsubseteq u$. Then $\mathcal{C}_{\bowtie}(u) = \mathcal{C}_{\bowtie}(v)$ and from Definition 4.10 we know $u = v$. □

Lemma 4.13
Let $(V, 0_V, \bowtie, \sqcup, \mathcal{T})$ be a transformable collision space without indistinguishable volumes. Then any two volumes have a unique infimum with respect to $\sqsubseteq$. ⌟

Proof. Let $u, v \in V$. As $\mathcal{C}_{\bowtie}(0_v) = \emptyset$ we know that $0_v \sqsubseteq u$ and $0_v \sqsubseteq v$, so a lower bound exists. Assume their infimum was not unique, so there are volumes x and y which are infima for u and v, and $x \neq y$. As there are no indistinguishable volumes, $\mathcal{C}_{\bowtie}(x) \neq \mathcal{C}_{\bowtie}(y)$ and as they are *greatest* lower bounds $x \not\sqsubseteq y$ and $y \not\sqsubseteq x$. From x and y being lower bounds, we know $\mathcal{C}_{\bowtie}(x) \subseteq \mathcal{C}_{\bowtie}(u) \cap \mathcal{C}_{\bowtie}(v)$ and $\mathcal{C}_{\bowtie}(y) \subseteq \mathcal{C}_{\bowtie}(u) \cap \mathcal{C}_{\bowtie}(v)$, thus $\mathcal{C}_{\bowtie}(x) \cup \mathcal{C}_{\bowtie}(y) \subseteq \mathcal{C}_{\bowtie}(u) \cap \mathcal{C}_{\bowtie}(v)$. But then $x \sqcup y \sqsubseteq u$ and $x \sqcup y \sqsubseteq v$, so $x \sqcup y$ is a lower bound of u and v, which is greater than both x and y as we established before that their collision sets each contain elements not in the collision set of the other volume. This contradiction concludes our proof. □

Corollary 4.14
Let $(V, 0_V, \bowtie, \sqcup, \mathcal{T})$ be a transformable collision space without indistinguishable volumes. Then $(V, \sqsubseteq)$ is a meet-semilattice[4] with least element 0_V. ⌟

The effect of over-approximation to selection (by collision) is shown by the next two lemmas.

Lemma 4.15
Let $(V, 0_V, \bowtie, \sqcup, \mathcal{T})$ be a transformable collision space and $s_1, \ldots, s_n, s'_1, \ldots, s'_n \in V$ with $s_i \sqsubseteq s'_i$ ($i \in \{1, \ldots n\}$). Then $\chi(\sigma_\bowtie(s_1, \ldots, s_n)) \subseteq \chi(\sigma_\bowtie(s'_1, \ldots, s'_n))$. ⌟

Proof. Let $v \in \chi(\sigma_\bowtie(s_1, \ldots, s_n))$. Then for each $i \in \{1, \ldots n\}$ it holds that $v \bowtie s_i$. With $s_i \sqsubseteq s'_i$ we known that also $v \bowtie s'_i$ and thus $v \in \chi(\sigma_\bowtie(s'_1, \ldots, s'_n))$. □

Lemma 4.16
Let $(V, 0_V, \bowtie, \sqcup, \mathcal{T})$ be a transformable collision space and $s_1, \ldots, s_n \in V$. Then for all $v, v' \in V$ with $v \sqsubseteq v'$ it holds that $\sigma_\bowtie(s'_1, \ldots, s'_n)(v) \implies \sigma_\bowtie(s'_1, \ldots, s'_n)(v')$. ⌟

Proof. Analogous to Lemma 4.15. □

Thus, over-approximating either the selection volumes or the selected volumes can only increase the number of volumes selected (analogously, under-approximation can only reduce the number of selected volumes). The next example demonstrates, that an increase in the number of selected volumes can take place.

Example 4.17
For a transformable collision space $(V, 0_V, \bowtie, \sqcup, \mathcal{T})$ let $s, v \in V$ with $\neg s \bowtie v$. Then, $\sigma_\bowtie(s)(v)$ does not hold, *i.e.*, s does not select v. Obviously, combining both volumes gives an over-approximation for each of them: $s \sqsubseteq s \sqcup v$ and $v \sqsubseteq s \sqcup v$. But as $s \bowtie s \sqcup v$ and $v \bowtie s \sqcup v$ hold, also $\sigma_\bowtie(s)(v \sqcup s)$ and $\sigma_\bowtie(s \sqcup v)(v)$ are true. ⌟

Limitations The structure presented so far, can express a relative position between volumes (parts in space). This includes the question whether two volumes collide or not, but also more complex relations, such as a distance defined by another volume acting as a *spacer*. For example,

$$\forall T \in \mathcal{T} : \ \neg(T(w) \bowtie u \wedge T(w) \bowtie v)$$

captures that some volume w never touches (collide with) both u and v at the same time, no matter where it is moved to. Thus, u and v are separated. However, the transformable collision space does not provide a measure of distance similar to a metric. Similarly, while the collision relation can be seen as a collision test, the intersection of two volumes can not be expressed. Although the infimum

[4] A *meet-semilattice* is defined as a partially ordered set where every non-empty set has an infimum.

can be seen as an approximation of intersection, there are spaces where the infimum is always the empty volume, *i.e.*, the volumes do not have to be closed under intersection.

These limitations are not problematic for our formal model of spatio-temporal systems (*c.f.*, Chapter 5) as we can model a large variety of real-world systems despite these limitations (*c.f.*, Chapter 8). However, we have to be aware of these restrictions when applying the model. For example distance sensors can not be modeled directly, but could be imitated by using three volumes describing the areas *near*, *medium*, and *far range* and checking for collision with them. In some cases this might even be favored, as it simplifies the model and ensures that measures from such a model (which always is a simplification of reality) are not interpreted as being exact, which can happen with real valued measurements.

4.2.3 Examples

To complete this section, we revisit some of the models of space from Section 4.2.1 to show how they can be embedded in the framework of transformable collision space.

Set Theory For a metric space with a universe of points U and metric $m : U \times U \to \mathbb{R}$ we can define a transformable collision space $(V, 0_V, \bowtie, \sqcup, \mathcal{T})$ as follows. Let V be the set of compact subsets of U, $0_V = \emptyset$, $\bowtie := \{(v_1, v_2) \in V \times V \mid v_1 \cap v_2 \neq \emptyset\}$, $\sqcup(v_1, v_2) := v_1 \cup v_2$, and $\mathcal{T}$ the set of functions $U \to U$ extended point-wise to subsets of U. It can be easily checked that all required axioms of Definition 4.1 are fulfilled. The requirements of a metric space and the choice of V as set of compact subsets is arbitrarily. This model based on set intersection and union can be easily transferred to other set systems as well.

Linear Algebra and CAD For the n-dimensional vector space $\mathbb{R}^n$ let P be the set of all fully dimensional polytopes. We define V as $\mathcal{P}_{\text{fin}}(P)$, *i.e.*, volumes are described by finite sets of multiple polytopes. We choose $0_v := \emptyset$, $\sqcup(v_1, v_2) := v_1 \cup v_2$, and

$$v_1 \bowtie v_2 \quad :\Leftrightarrow \quad \exists p_1 \in v_1, p_2 \in v_2 : \; p_1 \cap p_2 \text{ is fully dimensional} \; .$$

For the transformations we could for example use the set of affine transformations applied point-wise to the sets in V. Again, the axioms for a transformable collision space hold for our model.

A similar definition can be used to describe *constructive solid geometry* or other CAD space models in terms of a transformable collision space.

4.3 Formalizing Time

As we want to describe the behavior and spatial properties of a system over time, we also have to decide on a suitable formalization of time. When using the *usual* three dimensional Euclidean space, it might seem natural to just treat time as a fourth dimension. However, there are many reasons to treat time separately. First, we might need different levels of detail and thus different models for time and space. For example, we could use a simple coarse model for space while tracking time very precisely. Second, engineers and physicists typically tend to think about systems in dependence of time (at least in traditional mechanics). This can be seen in many of their laws, where quantities are interpreted as functions over time, and differentiation and integration are often performed with respect to the time variable. There are even special names for the derivatives, such as velocity and acceleration, for the first and second derivatives of location. Hence, treating time as a primary artifact and separating it from time both builds upon engineering/physics knowledge and can aid when discussing system models with colleagues of these disciplines. Finally, time does progress on its own even without external or internal actions occurring. This is so central, that modeling languages which allow the construction of models where time does not progress properly at some state[5] explicitly forbid these states and techniques for finding such states are developed.

The model we have chosen is based on linear time and streams and is introduced in the remainder of this section. We discuss the options available when using this time model and also touch some alternatives, but our goal is to agree on a simple model of time that is a sufficient basis for modeling spatio-temporal systems in the next chapter. While we explain, how more complicated models of time can be substituted, we use the simple time model to avoid overloading the spatio-temporal model with additional complexity from the time model. The contents of this section are mostly based on [Bro01, Bro08], but use a slightly different presentation for some of the details.

4.3.1 Linear Time

Although there are other models of time[6] we are using one-dimensional totally ordered time, *i.e.*, time is given by a pair $(\mathbb{T}, \leq)$, where $\mathbb{T}$ is a set of points in time and $\leq$ is a total order on $\mathbb{T}$. Typical instances of $\mathbb{T}$ are the natural numbers $\mathbb{N}$ and the non-negative real numbers $\mathbb{R}^{\geq 0}$ with the usual order relation. The first case is often called *discrete time*, the later *continuous* or *dense time*[7].

[5] More formally, these are states where an infinite amount of events occurs in a finite amount of time. Such behavior is usually referred to as *zeno* behavior, named after Zeno's paradox, which states that movement is not possible, as before another point is reached, you first have to cover half of the distance, and before that the quarter of the distance, and so on, leading to an infinite number of steps to perform.

[6] One example of a different time model is multi-dimensional time, which can be relevant when analyzing systems with multiple (drifting) local clocks.

[7] There is a subtle difference, as density only requires that between any two points in time there also is another time point, while for continuous time we would also require it to be a complete space (in the sense that the limit of each Cauchy sequence is also contained in the set). For example, $\mathbb{Q}$ would be dense but not continuous. In the context of time models, often these cases are not clearly differentiated, as the difference is not relevant for most models. This is also true in this thesis, where both terms are often used interchangeably. For both *continuous* and *dense* time the reader is advised to just think of $\mathbb{R}^{\geq 0}$.

We use the common notation of right-open intervals to refer to consecutive sets of time points; for $t_1, t_2 \in \mathbb{T}$ define

$$[t_1, t_2) := \{s \in \mathbb{T} \mid t_1 \leq s \wedge s < t_2\} \ .$$

Obviously, $[t, t) = \emptyset$.

Duration A typical question, given two points in time, is the amount of time passed between them. As time points are just *positions* in time, we need another structure describing these *time differences* or *durations* given by a tuple $(\mathbb{D}, 0, +, \leq)$, where $\mathbb{D}$ is the set of durations with binary addition $+$, such that $(\mathbb{D}, +)$ is a commutative monoid with identity element 0, and $\leq$ is a total order on $\mathbb{D}$. Additionally, $+$ has to be monotonic with respect to $\leq$, *i.e.*, for all $d_1, d_2, d_3, d_4 \in \mathbb{D}$

$$d_1 \leq d_2 \ \wedge \ d_3 \leq d_4 \quad \Longrightarrow \quad d_1 + d_3 \leq d_2 + d_4 \ .$$

To bridge the gap between time points $\mathbb{T}$ and durations $\mathbb{D}$ we require a partial function $\mathrm{dur} : \mathbb{T} \times \mathbb{T} \to \mathbb{D}$ which returns the duration between time points t_1 and t_2, if $t_1 \leq t_2$. The function has to obey $\mathrm{dur}(t, t) = 0$ for all points in time, and as we are dealing with one-dimensional linear time, we expect for all $t_1, t_2, t_3 \in \mathbb{T}$

$$t_1 \leq t_2 \leq t_3 \quad \Longrightarrow \quad \mathrm{dur}(t_1, t_2) + \mathrm{dur}(t_2, t_3) = \mathrm{dur}(t_1, t_3) \ .$$

We extend duration to time intervals $[t_1, t_2)$ by

$$\mathrm{dur}([t_1, t_2)) := \mathrm{dur}(t_1, \mathrm{sup}([t_1, t_2))) \ ,$$

where sup describes the supremum (also called least upper bound, or lub) of a set, *i.e.*, the smallest element which is an upper bound for all elements of the set.

Shifting For a given duration $d \in \mathbb{D}$ and time $s \in \mathbb{T}$, we call s *shiftable by* d, iff there is a time $t \in \mathbb{T}$ with $\mathrm{dur}(s, t) = d$. We then refer to t as $s + d$, and if defined, we also write $t - d$ for s. We call a pair of time $(\mathbb{T}, \leq)$ and durations $(\mathbb{D}, +, 0, \leq)$ with duration function dur *shifting compatible*, iff for all $t \in \mathbb{T}$ and $d \in \mathbb{D}$ we can shift t by d.

Example 4.18
Let $\mathbb{T} = \mathbb{N}$ and $\mathbb{D} = \mathbb{N}$ with the canonical operations for $\leq$ and $+$. Define $\mathrm{dur}_1, \mathrm{dur}_2 : \mathbb{T} \times \mathbb{T} \to \mathbb{D}$ by

$$\mathrm{dur}_1 : (t_1, t_2) \mapsto t_2 - t_1 \ \text{ and}$$

$$\mathrm{dur}_2 : (t_1, t_2) \mapsto \sum_{i=t_1+1}^{t_2} (i)^2 \ .$$

Both dur_1 and dur_2 are valid duration functions for $\mathbb{T}$ and $\mathbb{D}$, but $\mathbb{T}$ and $\mathbb{D}$ are shifting compatible only for dur_1, not for dur_2. We refer to the first case as *equidistant time*. ⌟

4.3.2 Streams

Streams are finite and infinite sequences which are interpreted as elements or events over time in this context. Let time be given by $(\mathbb{T}, \leq)$ with $\mathbb{T}$ being unbounded (*i.e.*, there is no largest element in $\mathbb{T}$ with respect to $\leq$), and M an arbitrary set. By $M_{\mathbb{T}}^{\infty}$ we denote the set of all mappings $\mathbb{T} \to M$, which we call *infinite streams*, and by $M_{\mathbb{T}}^{*}$ the set

$$\bigcup_{t \in \mathbb{T}} \{[0, t) \to M\} \ ,$$

which we call *finite streams*. Finite here refers to the finite duration of the stream given by $\text{dur}(0, t)$, as there can be an infinite number of elements in a single finite stream for dense time. For $\mathbb{T} = \mathbb{N}$ finite streams are equivalent to the *usual* finite sequences M^*. We denote all finite and infinite streams by $M_{\mathbb{T}}^{\omega} := M_{\mathbb{T}}^{*} \cup M_{\mathbb{T}}^{\infty}$.

The duration of a stream $x \in M_{\mathbb{T}}^{\omega}$ is written as $|x| \in \mathbb{D} \cup \{\infty\}$ and defined as

$$|x| = \begin{cases} \text{dur}(\text{dom}(x)) & \text{if } x \in M_{\mathbb{T}}^{*} \\ \infty & \text{if } x \in M_{\mathbb{T}}^{\infty} \end{cases} \ .$$

Concatenation of streams $x, y \in M_{\mathbb{T}}^{\omega}$ is only defined for shifting compatible time/duration combinations. In this case, we denote the concatenation by $x \frown y$ and define it by the following equations:

$$\begin{aligned} \text{dom}(x \frown y) &= \begin{cases} [0,\ 0 + (|x| + |y|)) & \text{if } x, y \in M_{\mathbb{T}}^{*} \\ \mathbb{T} & \text{otherwise} \end{cases} \\ (x \frown y).t &= \begin{cases} x.t & \text{if } t \in \text{dom}(x) \\ y.(t - |x|) & \text{otherwise} \end{cases} \end{aligned}$$

For streams $x, y \in M_{\mathbb{T}}^{\omega}$ denote the prefix relation by $x \sqsubseteq y$ and define it by

$$x \sqsubseteq y \quad :\Leftrightarrow \quad \text{dom}(x) \subseteq \text{dom}(y) \ \wedge \ y|_{\text{dom}(x)} = x \ .$$

If we have shifting compatible time and durations (*i.e.*, concatenation is defined), this is equivalent to the definition found in [Bro08]:

Corollary 4.19
Let $\mathbb{T}$ and $\mathbb{D}$ be defined as before, and shifting compatible; $\mathbb{T}$ unbounded. Then for all $x, y \in M_{\mathbb{T}}^{\omega}$

$$x \sqsubseteq y \ \Leftrightarrow \ \exists z \in M_{\mathbb{T}}^{\omega} : \ x \frown z = y \ .$$

⌟

For a stream x we denote its prefix until time $t \in \mathbb{T}$ by $x{\downarrow}t$, which is uniquely defined by

$$x{\downarrow}t \sqsubseteq x \quad \text{and} \quad |x{\downarrow}t| = \text{dur}(0, t) \ .$$

We also use the element-wise extension of this operator to sets and sequences of streams.

Streams and Time We use streams as a record of events and states over time. For example, a stream could be used to model all inputs to a system over the entire lifetime of the system. Depending on the time $\mathbb{T}$ chosen and the properties we are interested in, there are multiple ways of capturing these input events using a stream.

Assume the set of all possible inputs is given by I, then the most direct way of modeling inputs over time is to use a stream from $I^{\infty}_{\mathbb{T}}$. We choose an infinite stream here, as we assume the system to be running *for ever* as a simplification. So the stream captures for each time the input provided. For dense time we usually interpret the time to be the exact time the input event was observed, while for discrete time we interpret the input event to have occurred somewhere between this time point and the previous one. This simple model, however, has two flaws. First, it can not capture the absence of input at a certain time, and second it can not model the case of multiple input events for the same time. The first case can be addressed by introducing an explicit input ϵ, which is used to express the absence of input. For the second problem, there are two predominant solution.

One option is to simply ignore the problem. For dense time it is often even a valid assumption that no two input events may occur at the same time. For example all inputs might arrive over a common bus which technically does not allow multiple simultaneous messages. For discrete time this is different, as entire time intervals are considered. Again, we can sometimes rule out the presence of multiple input events, *e.g.*, if the sampling rate (*i.e.*, the duration between consecutive times) is lower than the rate at which inputs, possibly limited by bandwidth, occur. Alternatively, it can be defined that the stream captures the last (or first) event that occurred at the time span.

The second option is to consider streams $(M^*)^{\omega}_{\mathbb{T}}$, which capture for each time (interval) the finite and possibly empty sequence of inputs that where observed. For discrete streams we avoid losing input events this way, but the exact timing of events is lost. For events between consecutive time points, we then only know their relative order.

Stream Processing Functions Let $I_1, \ldots, I_n$ and $O_1, \ldots, O_m$ be sets of streams. A function

$$F : I_1 \times \ldots \times I_n \to \mathcal{P}(O_1 \times \ldots \times O_m)$$

is called a *stream processing function*. If we introduce $I := I_1 \times \ldots \times I_n$ and $O := O_1 \times \ldots \times O_m$ we can write more compactly $F : I \to \mathcal{P}(O)$. Instead of the power set construction we can also interpret a stream processing function as a relation $F \subseteq I \times O$. If F does not provide output for some input, *i.e.*, $\exists i \in I : \ F(i) = \emptyset$, F is called *partial*. If there is exactly one output for each input, *i.e.*, $\forall i \in I : \ |F(i)| = 1$, F can also be interpreted as a function $F : I \to O$.

4.4 Relating Time and Space

So far we only discussed time and space in isolation. To describe the behavior of spatial systems over time, however, we have to relate both quantities to each other. Our solution for this is to use

streams over volumes. For a fixed transformable collision space $(V, 0_V, \bowtie, \sqcup, \mathcal{T})$ and time domain $\mathbb{T}$ we describe the spatial extension of an element over time by a stream from $V_{\mathbb{T}}^{\omega}$. We call such a stream of volumes a *volume stream*.

The operators $\bowtie$ and $\sqcup$ can easily be extended point-wise to these streams ($v_1, v_2 \in V_{\mathbb{T}}^{\infty}$) by

$$(v_1 \bowtie v_2).t \;:=\; v_1.t \bowtie v_2.t \qquad \text{and} \qquad (v_1 \sqcup v_2).t \;:=\; v_1.t \sqcup v_2.t \;\;.$$

We limit the definition to infinite streams, but a generalization to both finite and infinite streams is possible by embedding the finite streams into the infinite ones by padding them with the empty volume, *i.e.*, $v_1 \frown V_0^{\infty}$.

In the definition above, $\bowtie$ is not a relation on streams, but rather $v_1 \bowtie v_2$ is a stream of Boolean values ($\mathbb{B}_{\mathbb{T}}^{\infty}$). We can obtain a relation on volume streams by aggregating this Boolean stream. For example, the relation $\bowtie_{\vee}$, which is defined as follows, captures whether the volumes of two elements collide at least once while time goes by:

$$v_1 \bowtie_{\vee} v_2 \;:=\; \bigvee_{t \in \mathbb{T}} (v_1 \bowtie v_2).t$$

On this time/space model we base our behavior of space-intensive systems, which is described in the next chapter. We will limit this discussion to infinite streams most of the time. This is to make the presentation more concise as the inclusion of finite streams often has to be handled as a special case. The embedding of finite streams into the infinite streams by appending a *null stream*, as shown above, still allows to carry over our model and the results to finite streams.

5 Modeling Spatio-Temporal Systems

This chapter introduces the semantic framework we are using for modeling spatio-temporal systems. The entire system itself is treated as a so called *spatio-temporal component* which may either be described directly (atomic component) or by composition of other components. This allows the decomposition of larger systems to manage their complexity, but also supports the creation of a library of basic building blocks which are used and reused in different models.

We start our presentation by recapitulating FOCUS, which is the foundation for our model. Then we describe a basic version of our model for spatio-temporal components, followed by a refined version which also supports dynamic addition and removal of components, which is required to model material. This model in both the basic and the extended version forms (with Chapter 6) the main contribution of this thesis. We then demonstrate the application of the modeling approach in an example (Section 5.4) followed by a discussion of the modeling theory with respect to extensions and limitations.

5.1 Software Systems: The FOCUS Approach

The starting point for our model of spatio-temporal systems is FOCUS [BS01]. Its core idea is to model only the input/output interface of a component or system, based on strongly causal stream processing functions (*c.f.*, Section 4.3.2). Here, we only summarize these parts of FOCUS, which we are reusing for our model. Furthermore, we limit our view to streams in M^∞, *i.e.*, we consider only systems which run infinitely long and *record* only one item for a time point (not a sequence of them).

In FOCUS, we differentiate between the syntactic and the semantic interface. The *syntactic interface* is given by communication endpoints and the (data) types which can be exchanged across these. The *semantic interface* describes, which exact outputs are to be expected for given inputs.

5.1.1 Types and Channels

To describe the syntactic interface of a component, we need the notion of a type. There is a plethora of works on type systems, which can grow fairly complex, including polymorphic types, recursive types, or inheritance between types. In our setting, however, we use a fairly simple type model which only takes the carrier set of a type into consideration. This is sufficient for describing the

component model, and general enough to be mapped to more elaborate type systems. We assume a set TYPES of type symbols and a universe DATA of all possible data values. By the function car : TYPES $\to \mathcal{P}(\text{DATA})$ we assign the set of valid data elements to a type (its *carrier set*).

The communication endpoints are called *channels* in FOCUS, and are labeled to simplify disambiguation. We call a set L of channel labels a *typed channel set*, if there is a mapping $\text{type}_L : L \to$ TYPES, which assigns to each channel its type. For a typed channel set L, a *channel valuation* is a type correct mapping from channels to streams, so the set of all channel valuations of L, denoted by $\vec{L}$, is given by

$$\{x : L \to \text{DATA}^\infty \mid \forall l \in L : \ x(l) \in \text{car}(\text{type}_L(l))^\infty\} \ .$$

We sometimes assume a linear order on channels and identify channel valuations with tuples of streams. By this interpretation we also apply the prefix operator $x \downarrow t$ to channel valuations in the canonical manner. Furthermore, as channel valuations are just functions, for any $C' \subseteq C$ and $c \in \vec{C}$ the limitation of c to channels from C' is given by $c|_{C'} \in \vec{C'}$.

5.1.2 Components and Composition

The syntactic interface of a component C in FOCUS is described by two typed channel sets I and O describing its input and output channels. The semantics of the component is provided by a stream processing function[1]

$$C : \vec{I} \to \mathcal{P}(\vec{O}) \ ,$$

i.e., input valuations (also called input histories) are mapped to all possible output valuations. The streams capture the flow of time, *i.e.*, they are interpreted as input and output values over time.

In FOCUS, the function C is required to be *total*, *i.e.*, produce at least one output history for each input history. The theory also can be extended to partial functions, which are then called *services* (*c.f.*, [BKM07]). We neglect this differentiation and allow the stream processing function of a component to be partial as well. An input history producing an empty set of output histories is considered an *invalid input*. If C provides at most one output history for each input history, *i.e.*, $\forall i \in \vec{I} : \ |C(i)| \leq 1$, we call it *deterministic*, otherwise it is called *nondeterministic*. Nondeterminism can be used to express uncertainty about the exact outcome of an operation, and is introduced either to capture under-specification or is the result of abstraction, *i.e.*, some input or effect which separates between various outputs has been omitted from the model.

[1] We tend to use the same name for the component and the function, although strictly speaking the stream processing function is a part (the semantics) of the component. However, from the context it is usually obvious which one is meant and using the same name helps in keeping the relation between components and functions, if multiple of them are involved.

Causality A crucial ingredient for any sound system model is a causal dependence between inputs and outputs. For components this is captured by the property of causality. A component C with syntactic interface given by (I, O) is called *(weakly) causal*, iff for all inputs $i_1, i_2 \in \vec{I}$ with $C(i_1) \neq \emptyset$ and $C(i_2) \neq \emptyset$ and any $t \in \mathbb{T}$

$$i_1{\downarrow}t = i_2{\downarrow}t \implies \{y{\downarrow}t \mid y \in C(i_1)\} = \{y{\downarrow}t \mid y \in C(i_2)\} \ . \tag{5.1}$$

This definition, adapted from [BKM07], states that for all valid inputs (those with non-empty output sets) the output until some time t is determined by the input until time t. If this would not hold, the output at some time t could depend on future input, *i.e.*, the component could predict the future, which is not consistent with how systems are usually perceived.

An even stronger requirement is strong causality, which in addition to weak causality assumes that *information processing takes time*, *i.e.*, output may not even depend on inputs from the same time, but there is a certain delay between inputs and dependent output. Formally, a component C with interface (I, O) is called *strongly causal*[2], iff there is a positive delay $\delta \in \mathbb{D}$, $\delta > 0$ such that for all inputs $i_1, i_2 \in \vec{I}$ with $C(i_1) \neq \emptyset$ and $C(i_2) \neq \emptyset$ and any $t \in \mathbb{T}$

$$i_1{\downarrow}t = i_2{\downarrow}t \implies \{y{\downarrow}\text{inc}(t,\delta) \mid y \in C(i_1)\} = \{y{\downarrow}\text{inc}(t,\delta) \mid y \in C(i_2)\} \ , \tag{5.2}$$

where $\text{inc}(t, \delta)$ is defined as

$$\text{inc}(t,\delta) := \min\{s \in \mathbb{T} \mid t \leq s \ \wedge \ \text{dur}(t,s) \geq \delta\} \ .$$

The minimum of the given set is trivially defined for discrete time and also exists for dense time, due to using $\geq \delta$. For shifting compatible time and duration, $\text{inc}(t, \delta)$ is simply $t + \delta$. The amount δ can be interpreted as the minimal reaction delay of a system or component. In the simple case of $\mathbb{T} = \mathbb{N}$ we can use the *successor* instead of a delay, which simplifies Equation 5.2 to

$$i_1{\downarrow}t = i_2{\downarrow}t \implies \{y{\downarrow}(t+1) \mid y \in C(i_1)\} = \{y{\downarrow}(t+1) \mid y \in C(i_2)\} \ . \tag{5.3}$$

Note that $t + 1$ returns the *first time point after* t, so 1 is not a duration here.

In FOCUS all components are required to be defined by strongly causal stream processing functions. Besides the assumption, that information processing takes time, this is also a prerequisite to obtain an expressive, yet simple and well-defined notion of composition.

Composition In FOCUS a single composition operator is used, from which the usual triad of sequential composition, parallel composition, and feedback can be obtained as special cases. Furthermore, following the introduction in [Bro08], the composition is only based on the names of the channels. For components C_1, C_2 with interfaces (I_1, O_1) and (I_2, O_2) such that $I_1 \cap O_1 = I_2 \cap O_2 = O_1 \cap O_2 = \emptyset$, their *composition* $C_1 \otimes C_2$ is defined as follows. The syntactic interface

[2] This property is also referred to as *delayed behavior* [MS97] or *time guarded by a finite delay* [Bro01] in the literature.

of $C_1 \otimes C_2$ is given by (I, O) with $I = (I_1 \setminus O_2) \cup (I_2 \setminus O_1)$ and $O = O_1 \cup O_2$, *i.e.*, all inputs for which an output with the same name exists are *connected* to them and thus removed from the component's interface. The semantics is provided by the following equation for all $i \in \vec{I}$:

$$(C_1 \otimes C_2)(i) = \left\{ o \in \vec{O} \mid \exists z \in \vec{Z} : \; z|_I = i \; \wedge \; o|_{O_1} \in C_1(z|_{I_1}) \; \wedge \; o|_{O_2} \in C_2(z|_{I_2}) \right\} \quad , \quad (5.4)$$

where $Z := I_1 \cup I_2$ is the set of all input channels of both components. This enforces the connected inputs and outputs to have the same histories, *i.e.*, the same values at the same time. As the definition of composition contains recursion, it is only sensible if we can guarantee a unique solution for this equation. It turns out, that only if both C_1 and C_2 are strongly causal, $C_1 \otimes C_2$ is well-defined and strongly causal as well. Intuitively, for $\mathbb{T} = \mathbb{N}$, this is given by the fact that the output at time t is already defined by the inputs up to time $t-1$ (thanks to strong causality), from which then the input for time t and thus the output for time $t+1$ can be obtained. By induction, all valid output histories can be constructed. The formal proof in the discrete time case is usually based on the Knaster-Tarsky theorem, which guarantees the existence of a least fixed point and thus a solution. The more general proof, which also applies to dense time, involves *contractive functions* (which happen to be related to strongly causal stream processing functions) and Banach's fixed point theorem and can be found in [MS97].

5.1.3 Time Synchrony versus Time Asynchrony

System models which do not require a delay for either information processing or information passing between components are usually called time synchronous. Strong causality for components is relaxed to weak causality there. These models allow a component's output to depend on input read at the same time. Prominent instances of synchronous system models are Esterel and Lustre [BCE+03, Ber07] and the proprietary modeling language and tool MATLAB/Simulink. In contrast, the asynchronous system models, such as FOCUS, require a small delay in either information processing or communication[3]. Both the synchronous and asynchronous approaches have advantages and disadvantages, which we summarize shortly.

Well-Definedness of Composition In synchronous approaches the construction of so called *causal loops* is possible. These are chains of components, such that for each component its output depends on input delivered by the previous component in the chain. By closing the chain, *i.e.*, connecting the outputs of the last component with the inputs of the first one, there is a data path with circularly dependent values *at the same point in time*. If causal loops exist, the result of the composition is not always defined – a simple example of such a case is given in [Bro08]. As a result of this fundamental problem, lots of efforts in the area of synchronous languages have been invested in the construction of efficient algorithms for checking the existence of a unique fixed point (and

[3] In our presentation we enforce an information processing delay by strong causality, but the entire theory can be built as well on communication delay.

thus a valid composition) and its calculation in the presence of causal loops. A pragmatic solution to this problem, which is employed by tools such as Simulink, is to simply disallow causal loops. All circular dependencies have to be broken by a special delay element, which corresponds to a strongly causal buffer. With FOCUS, in contrast, causal loops can not occur, as each component is required to be strongly causal. This significantly simplifies the formal treatment of the models and especially the extension of the system model with new elements (*e.g.*, for spatio-temporal systems), which is the main reason for using FOCUS as the basis for our system model.

Accumulation of Delays One major drawback of FOCUS is the accumulation of delays when composing components. When performing a sequential composition of n components with a delay of δ each, their composition has a delay of $n\delta$. This is good, as is makes the composition strongly causal, but also means that certain components with low delay can not be decomposed any further without violating timing constraints. The effect becomes more severe with coarser time scales, that is discrete time with large durations between consecutive times. This problem does not exist for synchronous models, as the delay is 0 and thus can not accumulate.

Besides simply ignoring this problem, there are two predominant solutions. The first one is to allow *time scale refinement*. This means that a component may run on a more *precise clock*[4] internally, which allows it to compensate for any delays caused by composition. A thorough discussion of this solution and the mathematical framework are provided by [Bro08]. The other approach is the controlled integration of weakly causal components. Here both strongly and weakly causal components are allowed, as long as no loops of only weakly causal components are created. While there are cases, where this does not solve the problem of finding a suitable decomposition which respects timing, this is often sufficient for practical purposes. Here, we ignore the problem for this chapter to make the semantic foundation of the modeling theory easier to understand, but revisit it in Section 6.1.3, where the second solution is pursued.

The Physical World Whether the physical world appears synchronous or asynchronous depends on the time scale. The reaction of an electronic circuit on its inputs or the transmission of momentum from one billiard ball to another upon collision seem to be instantaneous and thus synchronous at first. When refining the time scale and looking more precisely, the electronic circuit requires a certain setting time until the results are available, and there is a small time span where both billiard balls are perfectly still. For some effects, such as gravity, the standard models of physics do not yet provide a definitive answer on whether the reactions are rather synchronous or asynchronous. Ultimately, the answer to whether our world is synchronous or asynchronous at its innermost has philosophical value only. As our intent is the creation of models, which are abstractions, there is always some error over the original. It is just important to be aware of the possible errors, independent of which approach is used, and to respect them when analyzing the model and interpreting any results.

[4] For discrete time this means a different discrete time with shorter durations, for dense time, this indicates a lower required delay for components.

5.2 Spatio-Temporal Components

This section introduces the foundation for our model of spatio-temporal systems. We try to keep the presentation similar to that from the previous chapter, but as our composition is slightly more involved, we present the parallel composition and feedback operation separately. The sequential composition and the more complex composition operation provided for FOCUS can be trivially constructed from these two operators.

From now on, we assume to have fixed an infinite discrete time (more concretely we use $\mathbb{T} = \mathbb{N}$) and a transformable collision space $(V, 0_V, \bowtie, \sqcup, \mathcal{T})$. Mechatronic systems contain a large amount of control algorithms, which seem to require a continuous model of time. But on the level of abstraction we are mostly interested in, these algorithms only play a minor role and can be abstracted by discrete processes most of the time. Thus, we decided to use the slightly more simple model of discrete time, where time advances in equidistant ticks and events within a single tick can not be differentiated. Furthermore, components can send and receive at most one message on each channel in a single tick. In our experience, most effects can be modeled and analyzed in this time model, as long as the length of a single time interval (tick) is chosen sufficiently small.

The limitation to this discrete time model simplifies both the understandability and analyzability as well as the creation of models and supporting tools. Many invariants are easier to describe by induction in a discrete time setting, than by the notations required for continuous time. So, more care can be taken of the interplay of behavioral and spatial properties, which in our opinion is the central problem on a more abstract modeling level. However, a generalization of the theory to more complex models of real or continuous time can be achieved following [Bro01] and is discussed in Section 5.5.1.

5.2.1 Component Interfaces

A system is a spatio-temporal component, which is described by its interface. While interfaces for software systems have been studied for years and are well understood, a similar notion for mechatronic or spatio-temporal systems does not exist. Although engineers sometimes use the term *interface* to refer to details of the electrical wiring, such as the number of wires or the shape of plugs used, this is not the notion of interface used in this model. Here, the interface of a component fully defines the possible interactions with the component and all possible observable reactions. This notion of *observability* is crucial, as we do not differentiate internal states if they can not be distinguished by external observation.

Our view of a spatio-temporal interface consists of four main parts. The first one is the *logical interface* (*i.e.*, input and output channels), which we already know from FOCUS. This is used for the same purposes as in pure software systems, *i.e.*, message exchange, but also as an abstract means to exchange physical units. For example, a gearbox can be abstracted by a component with

one input channel on which a pair of torque and revolutions per minute is received continuously, and a modified pair of these values is sent via the output.

The most obvious observable element of a spatio-temporal component is its shape or the space it allocates. We model this by so called *parts*, which are identifiers for volumes. A component may consist of multiple parts, which allows a more or less fine-grained description of a component. An abstract view might treat the component as consisting of a single part, while a more detailed treatment could require each individual screw to be represented by its own part. Generally, parts model the rigid elements, such as steel frames or the housing of sensors.

An important aspect of spatio-temporal systems is that they not only possess a spatial representation, but also react to changes in the surrounding space. A single component does not have a global view of space, but rather perceives space through certain sensors (not only meant in the technical sense), which monitor a limited area in space. We call these sensing elements *detectors* and they are identifiers for volumes that detect the presence of other parts. This detection is expressed by a collision between the volumes of the part and the detector. For mechatronic systems, detectors correspond to the working area of sensors, *e.g.*, the light ray of a photoelectric barrier or the area covered by a proximity sensor.

The last effect that can be observed at the boundary of a component is that the position of a component can be affected by another component. For example, an industrial robot consists of several segments which are attached to each other. Each of these segments can be interpreted as a single component which, by an electric drive, can move the component representing the attached segment. We refer to these attachment points as *movers*, *i.e.*, other components can be connected to these movers and their position is then affected by the component and the state of its mover[5]. Typically, movers represent actuators, such as pneumatic cylinders or electric drives.

Following FOCUS, we distinguish the syntactic interface, which describes the static aspects of a component (which channels and parts are present) and the semantic or behavioral interface (what reactions are expected for given stimuli). The interface description of a component contains all details required to use it and analyze its behavior, but does not disclose any details on how this behavior is achieved (implementation). Before providing the full definition, we discuss the properties required.

The syntactic interface of a spatio-temporal component C is described by a tuple (I, O, D, P, M), with typed channel sets I and O, and sets of detectors D, parts P, and movers M. All of these sets can be interpreted as labels used to identify the respective items and are expected to be pairwise disjunct.

Having fixed a transformable collision space $(V, 0_V, \bowtie, \sqcup, \mathcal{T})$, the semantics of C is defined by a

[5] Whether a system is modeled by a single component with multiple parts or multiple components connected by movers is a design decision to be taken by the modeler.

function[6]

$$F : \vec{I} \times \mathcal{A}(D) \to \mathcal{P}(\vec{O} \times \vec{D} \times \vec{P} \times \vec{M}) \ ,$$

where

- $\vec{I}$ and $\vec{O}$ are channel valuations as introduced in FOCUS,
- $\mathcal{A}(D) := \{D \to \mathbb{B}^\infty\}$ are detector activations,
- $\vec{D} := \{D \to V^\infty\}$ are detector positions/volumes,
- $\vec{P} := \{P \to V^\infty\}$ are part positions/volumes, and
- $\vec{M} := \{M \to \mathcal{T}^\infty\}$ are transformations exercised by movers.

Intuitively a spatio-temporal component receives data input and reads spatial information via its detectors ($\mathcal{A}(D)$). As a response it outputs data, changes the positions of its detectors and rigid parts, and modifies the transformations caused by its movers. The interplay between all of these will become clearer in the remainder of this section, when we discuss invariants and composition operators.

As we are interested in the behavior of the system over a possibly infinite time, all inputs and outputs are not single elements but streams or sets of streams. These are interpreted under the assumption of a discrete time, and each time interval contains exactly one element (channel value, position, etc.). This assumption is valid if the time steps are small enough, otherwise it can be interpreted as an abstraction (*c.f.*, Section 4.3.2). The power set is used again to both model partiality ($F(i, a) = \emptyset$) and nondeterminism ($|F(i, a)| > 1$) as a result of under-specification or abstraction of physical phenomena. Partiality is crucial for spatio-temporal components, as not all inputs may lead to a valid configuration of the system. For example, certain inputs could drive two robots through each other, leading to a collision and destruction of the robots for the real system. So, partiality here does not model missing specification, but rather is used to capture all inputs which potentially lead to disastrous results, which puts the term *chaos completion* to a different perspective.

We already explained in Section 5.1.3 why we are using an asynchronous model of computation. Consequently, we require spatio-temporal components to be strongly causal. The definition is basically the same as Equation 5.3, but we repeat it here in terms of the extended interfaces just introduced.

Definition 5.1
Let (I, O, D, P, M) be given as before. The function $F : \vec{I} \times \mathcal{A}(D) \to \mathcal{P}(\vec{O} \times \vec{D} \times \vec{P} \times \vec{M})$ is called *strongly causal*, iff for all inputs $(i_1, a_1), (i_2, a_2) \in \vec{I} \times \mathcal{A}(D)$ with $F(i_1, a_1) \neq \emptyset$ and $F(i_2, a_2) \neq \emptyset$ and any $t \in \mathbb{N}$

$$(i_1, a_1)\downarrow t = (i_2, a_2)\downarrow t \implies \{y\downarrow(t+1) \mid y \in F(i_1, a_1)\} = \{y\downarrow(t+1) \mid y \in F(i_2, a_2)\}$$

⌟

[6] In comparison to the definition from [Hum09] the notion of a *forbidden region* was dropped, as some equations become easier while no expressiveness is lost.

This allows us to capture the definition of a spatio-temporal component formally:

Definition 5.2 [Spatio-Temporal Component]
Let (I, O, D, P, M) be given as before, and F a strongly causal function $F : \vec{I} \times \mathcal{A}(D) \to \mathcal{P}(\vec{O} \times \vec{D} \times \vec{P} \times \vec{M})$. Then C described by the pair of both is called a *spatio-temporal component*, iff the following invariants hold for all $(i, a) \in \vec{I} \times \mathcal{A}(D)$ and $(o, d, p, m) \in F(i, a)$:

$$\forall t \in \mathbb{N}\, \forall \delta \in D :\ a(\delta).t \iff \bigvee_{\rho \in P} d(\delta).t \bowtie p(\rho).t \tag{5.5}$$

$$\forall t \in \mathbb{N}\, \forall \rho_1, \rho_2 \in P :\ p(\rho_1).t \bowtie p(\rho_2).t \implies \rho_1 = \rho_2 \tag{5.6}$$

The tuple (I, O, D, P, M) is called its *syntactic interface*, the function F its *semantic interface* or its *behavior*[7]. ⌟

Invariants 5.5 and 5.6 capture assumptions about the (physical) world, namely that detectors work faithfully, *i.e.*, if a *local* part is present, it is reported correctly, and different solid parts never overlap (collide). The implication for detector activations is not an equivalence, as there might be other parts in the world, which can trigger an activation of the detectors as well (open world assumption). We encounter more of these *world assumptions* for the extended model in Section 5.3. Also note that both sides of the implication in Equation 5.5 refer to the same time t, so detected parts are *reported* immediately by the world. This could lead to undefined situations, where a component changes the position of some parts based on a lack of activation, which in turn causes the parts to be in the range of the detector and thus enforce activation, contradicting the original input. The solution to this dilemma is once again provided by strong causality, which forbids the position at time t to depend on activations at time t.

Our approach is conservative in that the definition already rules out components which are physically impossible (according to our simplified laws of physics, *i.e.*, Invariant 5.6). All operators for modifying and constructing components from other components, which are described in the following sections, thus have to respect these invariants. Components built by application of these operators are *valid by construction* regarding our simplified physics. An alternative approach is to be less restrictive in the definition of a spatio-temporal component and the composition operators, but rule out violating components *after* construction/composition.

5.2.2 Parallel Composition

We begin our list of operations on components with the parallel composition, which corresponds to the case of putting both components next to each other. In terms of data exchange this is a trivial operation, but on the spatial level there are interactions between the parts and detectors of the

[7] As with FOCUS, we often use the same name for the component and its behavior function (which was referred to by F here), as it is usually clear from the context which is meant.

composed components. Volumes of the parts may activate the detectors of the other component, or parts might collide, leading to an invalid state (*i.e.*, an empty set of outputs for certain inputs).

Definition 5.3 [Parallel Composition of Spatio-Temporal Components]
For $j = 1, 2$ let C_j be spatio-temporal components with syntactic interface $(I_j, O_j, D_j, P_j, M_j)$ and semantic interface $C_j : \vec{I_j} \times \mathcal{A}(D_j) \to \mathcal{P}(\vec{O_j} \times \vec{D_j} \times \vec{P_j} \times \vec{M_j})$ such that their interfaces are disjunct, *i.e.*, $X_1 \cap Y_2 = \emptyset$ where X, Y are just substitutes for each of I, O, D, P, M. Their *parallel composition*, denoted by $C_1 \| C_2$, has the syntactic interface

$$(I, O, D, P, M) := (I_1 \cup I_2, O_1 \cup O_2, D_1 \cup D_2, P_1 \cup P_2, M_1 \cup M_2)$$

and a semantic interface F that is defined for an input $(i, a) \in \vec{I} \times \mathcal{A}(D)$ by

$$F(i, a) := \begin{cases} \emptyset & \text{if coll\_input}(i, a) \\ R(i, a) & \text{otherwise} \end{cases},$$

where

$$\begin{aligned} \text{coll\_input}(i, a) \ :\Leftrightarrow \ & \exists(\cdot, \cdot, p_1, \cdot, \cdot) \in C_1(i|_{I_1}, a|_{D_1}) \ \exists(\cdot, \cdot, p_2, \cdot, \cdot) \in C_2(i|_{I_2}, a|_{D_2}) \\ & \exists t \in \mathbb{N} \ \exists \rho_1 \in P_1, \rho_2 \in P_2 : \ p_1(\rho_1).t \bowtie p_2(\rho_2).t \end{aligned} \tag{5.7}$$

and $R(i, a)$ is the set of all $(o, d, p, m) \in \vec{O} \times \vec{D} \times \vec{P} \times \vec{M}$ for which the following two equations hold:

$$\forall j \in \{1, 2\} : \ (o|_{O_j}, d|_{D_j}, p|_{P_j}, m|_{M_j}) \in C_j(i|_{I_j}, a|_{D_j}) \tag{5.8}$$

$$\forall j, k \in \{1, 2\}, j \neq k \ \forall t \in \mathbb{N} \ \forall \delta \in D_j : \ a(\delta).t \iff \bigvee_{\rho \in P_k} p_k(\rho).t \bowtie d_j(\delta).t \tag{5.9}$$

⌟

Equation 5.8 ensures that the input and output, the positions of detectors and parts, and the mover transformations are just evaluated individually by each of the parallel composed components and Equation 5.9 describes that detector activations can be triggered from both the outside of the new component and mutually by the two components, *i.e.*, if a part from one component is detected by (collides with) a detector from the other component. Equation 5.7 limits composition to inputs without collisions between parts of the two components. More precisely, we *exclude* an input if there is a single possible pair of outputs of the composed components which would lead to an invalid state (collision). This is due to our nondeterministic interpretation of the behavior function which does not allow us to rule out the occurrence of exactly this output pair for the critical input. These equations together enforce the Invariants 5.5 and 5.6 (our world assumptions) and thus ensure that parallel composition is a well-defined operation on spatio-temporal components.

Lemma 5.4
Let C_1, C_2 be spatio-temporal components. Then their parallel composition $C_1 \| C_2$ is a spatio-temporal component as well. ⌟

Proof. As the syntactical construction is correct, the proof consists of checking three properties. The first one is strong causality, which is obviously fulfilled, as the new behavior function is constructed by parallel composition of two strongly causal behavior functions, and discarding some function results. Invariant 5.5 is fulfilled as all cases are either covered by the same invariant on the input components, or by Equation 5.9. The same holds for Invariant 5.6 with respect to Equation 5.7. □

From our definition it can be easily checked that the following statement holds.

Corollary 5.5
Parallel composition of spatio-temporal components as defined in Definition 5.3 is both a commutative and associative operation. ⌟

5.2.3 Data Feedback

With only parallel composition the components could only interact physically (*i.e.*, by collisions between parts and detectors). To support communication using the data input and output channels, we introduce a feedback operation.

Definition 5.6 [Feedback Operator]
Let C be a spatio-temporal component with syntactic interface (I, O, D, P, M) and semantic interface F, and $q : I \to O$ a *partial* mapping from input to output channels which is *type correct*, *i.e.*, $\forall \iota \in \text{dom}(q) : \text{type}_I(\iota) = \text{type}_O(q(\iota))$.

Then the *feedback on C according to q* is denoted by $\text{fb}(C, q)$ and has the syntactic interface (I', O, D, P, M) with $I' := I \setminus \text{dom}(q)$. For its behavior function F' for each input $(i', a) \in \vec{I'} \times \mathcal{A}(D)$ a tuple $(o, d, p, m) \in \vec{O} \times \vec{D} \times \vec{P} \times \vec{M}$ is in $F'(i, a)$, iff

$$\exists i \in I : \; i|_{I'} = i' \; \wedge \; (o, d, p, m) \in C(i, a) \; \wedge \forall \iota \in \text{dom}(q) : \; i(\iota) = o(q(\iota)) \; .$$

⌟

Due to the strong causality we required for spatio-temporal components, this operation is well defined and the function defined by a component after feedback can be easily constructed by induction on time.

Lemma 5.7
Let C be a spatio-temporal component, q a matching *type correct input/output mapping for C. Then $fb(C, q)$ is a spatio-temporal component.* ⌟

Proof. The critical part of the proof is to check that above recursive equation is well defined (a unique solution exists) and the resulting function is still strongly causal. However, the proof is basically the same as for the composition in FOCUS, as explained in Section 5.1.2 and thus omitted here. As feedback is only making the domain and image of the behavior function smaller, the validity of Invariants 5.5 and 5.6 is trivially preserved. □

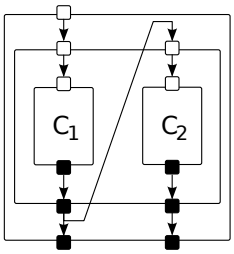


Figure 5.1: Graphical definition of sequential composition.

Feedback is usually used in conjunction with parallel composition to connect multiple components with each other. A special case which can be achieved by combination of parallel composition and feedback is sequential composition, where components are *chained* behind each other. The construction for sequential composition by parallel composition and feedback is sketched in Figure 5.1. Parallel composition and feedback are also sufficient to emulate the more general composition operator from FOCUS (*c.f.*, Section 5.1.2).

What is interesting about the two operations introduced so far is that they either only affect the spatial parts (parallel composition) or the logic communication (feedback). This is true for the operations presented in the remainder as well. Thus, the interplay between logic communication and spatial effects (motion and collisions) has to happen in the atomic components. An example for this is given in Section 5.4, where a sensor is modeled as a component which converts detector activations to data sent via the output channel.

5.2.4 Positioning and Connecting Movers

An operation that seems obvious when dealing with components in space is a positioning operation, which is used to move a component to its proper place. As we are working with relative transformations over time, we have to integrate them to yield an absolute transformation.

Definition 5.8 [Integrated Transition Stream]
For a stream $T \in \mathcal{T}^\infty$ of transformations, its *integrated transition stream* is denoted by $\text{its}(T)$ and defined inductively by $\text{its}(T).0 = T.0$ and $\text{its}(T).(i+1) = T.(i+1) \circ \text{its}(T).i$. ⌟

We use a generalized version of a positioning operator, which applies an incremental transformation to the spatial parts of the component at any point in time.

Definition 5.9 [Positioning Operator]
Let C be a spatio-temporal component with syntactic interface (I, O, D, P, M), semantic interface F, and $T \in \mathcal{T}^\infty$ a stream of transformations. Then $\text{pos}(C, T)$ denotes C *positioned by* T. The positioned component has the same syntactic interface as C and its semantic interface F' is defined as follows. For a pair $(i, a) \in \vec{I} \times \mathcal{A}(D)$ the tuple $(o, d', p', m') \in F'(i, a)$, iff there are $d \in \vec{D}$,

$p \in \vec{P}$, and $m \in \vec{M}$ such that $(o, d, p, m) \in F(i, a)$ and the following equations hold for each $t \in \mathbb{N}$:

$$\forall \delta \in D : \; d'(\delta).t = (\text{its}(T).t)(d(\delta).t) \tag{5.10}$$

$$\forall \rho \in P : \; p'(\rho).t = (\text{its}(T).t)(p(\rho).t) \tag{5.11}$$

$$\forall \mu \in M : \; m'(\mu).t = (T.t) \circ (m(\mu).t) \tag{5.12}$$

⌟

Equations 5.10 and 5.11 describe that all positions of detectors and parts are just transformed to the new (absolute) position. Equation 5.12 captures the property that a component connected to a mover (which is explained in more detail below) will not only be affected by the transformation of the mover, but also by transformations to the component itself. As the mover also performs incremental transformations, contrary to the parts and detectors no integration is required.

Remark 5.10
For a transformation $\tau \in \mathcal{T}$, we will also use $\text{pos}(C, \tau)$ as an abbreviation for $\text{pos}(C, \tau \frown 0^\infty)$, where $\tau \frown 0^\infty$ denotes the stream consisting of an initial transformation of τ followed by 0s, *i.e.*, no further transformations. This captures the common case of an initial but fixed placement. ⌟

As all volumes of the component are affected by the same transformation, and transformations preserve collisions (Equation 4.4), the following result is easily derived.

Corollary 5.11
Let C be a spatio-temporal component, $T \in \mathcal{T}^\infty$ a stream of transformations. Then pos(C, T) is a spatio-temporal component. ⌟

Furthermore, positioning a component has no impact on the validity with respect to our simplified laws of physics, *i.e.*, the partiality of the component's behavior function does not change, according to the definition.

Corollary 5.12
Let C be a spatio-temporal component with syntactic interface (I, O, D, P, M), semantic interface F, $T \in \mathcal{T}^\infty$ a stream of transformations, and the semantic interface of pos(C, T) given by F'. Then

$$\forall i \in \vec{I}, a \in \mathcal{A}(D) : \; |F(i, a)| = |F'(i, a)| \; .$$

⌟

Having defined positioning, we finally can express that a component is connected to another one (like for example a robotic gripper is connected to a robotic arm) via some mover and thus has to follow every movement.

Definition 5.13 [Connection to a Mover]
For $j = 1, 2$ let C_j be spatio-temporal components with syntactic interface $(I_j, O_j, D_j, P_j, M_j)$ and $\mu \in M_1$ a mover. Then the *connection of C_2 to C_1 via the mover μ*, denoted by $C_1 \overset{\mu}{\circ\!\!-} C_2$, is defined by

$$C_1 \overset{\mu}{\circ\!\!-} C_2 \ := \ C_1 \parallel \text{pos}(C_2, \text{its}(\pi_\mu(C_1))) \ ,$$

where π_μ is the projection of the behavior function to the single output corresponding to the transformation of μ. ⌟

As a connection by a mover results in a single component, no cyclic connections can occur. Thus the connections by movers form a forest in the components, which corresponds to serial kinematics (also called serial chains).

5.2.5 Compatibility with FOCUS

The model described in this section has been introduced as an extension of the FOCUS theory to spatio-temporal systems. In the next paragraphs we formalize this fact. More concrete, we show that spatio-temporal components without spatial aspects are isomorphic to FOCUS components.

Definition 5.14
A spatio-temporal component C with syntactic interfaces (I, O, D, P, M) is called *space-less*, iff $D = P = M = \emptyset$. ⌟

For the remainder, we use the set $\mathcal{C}_{\text{FOCUS}}$ to denote all possible FOCUS components, and $\mathcal{C}_{stsl}$ for all space-less spatio-temporal components.

Definition 5.15
The bijection $\iota : \mathcal{C}_{\text{FOCUS}} \to \mathcal{C}_{stsl}$ which maps a FOCUS component C with syntactic interface (I, O) and semantic interface F to the spatio-temporal component C' with syntactic interface $(I, O, \emptyset, \emptyset, \emptyset)$ and semantic interface F is called the *canonical embedding of* FOCUS . ⌟

To define an isomorphism, we have to define a composition operator for spatio-temporal components, which is similar to the one we introduced for FOCUS.

Definition 5.16
For $j = 1, 2$ let C_j be spatio-temporal components with syntactic interface $(I_j, O_j, D_j, P_j, M_j)$ and semantic interface $C_j : \vec{I_j} \times \mathcal{A}(D_j) \to \mathcal{P}(\vec{O_j} \times \vec{D_j} \times \vec{P_j} \times \vec{M_j})$ such that their interfaces are disjunct with the exception that (I_1, O_2) and (I_2, O_1) may overlap. Their *naming composition*, denoted by $C_1 \tilde{\otimes} C_2$ is defined as

$$C_1 \tilde{\otimes} C_2 := \text{unrename}(\text{fb}(q, \text{rename}(C_1) \| \text{rename}(C_2))) \ ,$$

where rename is a functions which renames the channels in C_1 and C_2 to non-colliding names, unrename the inverse function, and q a partial mapping from renamed inputs to renamed outputs, such that channels, which had the same name before renaming, are connected. ⌟

This composition operator, constructed from parallel composition and feedback, allows the definition of a substructure of spatio-temporal components which is isomorphic to FOCUS. This is captured in the following proposition, which is left without proof, as the proof is not interesting in technical terms, but requires a significant notational overhead.

Proposition 5.17
The canonical embedding of FOCUS ι *is an isomorphism from* $(\mathcal{C}_{\text{FOCUS}}, \otimes)$ *to* $(\mathcal{C}_{stsl}, \tilde{\otimes})$. ⌟

This result is relevant in two different ways. First, it allows us to transfer results and techniques from the FOCUS world (at least the slightly restricted one presented here) to our model, as long as we do not include spatial properties in our components. This is also relevant when working towards tooling, as ideas and parts of tools for FOCUS can be reused. As a result, the tool prototype presented in Chapter 7 shares a large common code base with a tool based on FOCUS. Second, this result shows that our intention of creating a spatio-temporal extension of FOCUS has been successful. Thus, in terms of software systems the model should be as faithful in capturing real systems as FOCUS. For the spatial part, this has still to be discussed, which is deferred to Section 5.5.

5.3 Dynamic Component Generation: Dealing with Material

As indicated in the introduction, the domain we concentrate on, is automation engineering. The model presented so far can capture many properties of mechatronic systems. However, it still lacks the ability to describe the primary subject of automation engineering: material handling and processing. Introducing material to the model has a couple of consequences:

- Material has to be represented in the model and its interaction with the remaining system and possibly other material has to be described.
- Typically, material is not just present during the entire life time of the system, but is inserted from some external system or operator, and removed (usually by the system itself) by moving it *out of scope*.
- As material is handled by the system, its position should be affected by movers. This requires dynamic attachment of movers to material.
- Collisions are not necessarily an error which has to be avoided (by partiality of behavior), instead they are actively used to stop the motion of material. So, motion implied to other parts may be ignored in some cases.

There are multiple possibilities to deal with these aspects in our model. Here we only present one of these solutions, which evolved over several years and is heavily influenced by our experience in practical application of the model (*c.f.*, Chapters 6, 7 and 8).

For modeling material we just use spatio-temporal components as described before. The major advantage is that we do not have to introduce new modeling primitives and can reuse all composition operators defined so far. Furthermore, this allows the description of fairly complex material. An example could be a system that assembles toy robots. These robots are transported and assembled by the system and are thus treated as material. However, the robots are small systems themselves and might perform complex motions based on environment observations while being transported from one subsystem to the next.

The aspects of material injection and extraction as well as the dynamic attachment to material are handled by extended spatio-temporal components, which are introduced next. The extension of the component interface also leads to extended world assumptions and semantics.

5.3.1 Extended Spatio-Temporal Components

Based on Definition 5.2 we extend the interface of spatio-temporal components by a set of *entries* Y, *exits* X[8], and *bindings* B. All of these are finite sets containing labels used for identification of the respective parts of the model. The entries describe where and when material may enter the scope of the modeled system, while exits describe where and when material will leave this scope. Bindings describe in which case and how material is bound to the interface of the systems, *i.e.*, its inputs, outputs, and movers. So, for an extended spatio-temporal component C, the syntactic interface is defined by the tuple (I, O, D, P, M, B, Y, X).

For the semantics of the component, we use spatial selection predicates as introduced on Section 4.2.2, and especially the set $\mathcal{S}$ of all spatial selection predicates. The semantics of an extended spatio-temporal component C is defined by a function

$$F : \vec{I} \times \mathcal{A}(D) \to \mathcal{P}(\vec{O} \times \vec{D} \times \vec{P} \times \vec{M} \times \vec{B} \times \vec{Y} \times \vec{X}) \ ,$$

where

- $\vec{I}$, $\mathcal{A}(D)$, $\vec{O}$, $\vec{D}$, $\vec{P}$, and $\vec{M}$ are the same as in Definition 5.2,
- $\vec{B} := \{B \to (\mathcal{S} \times (I \cup O \cup M))^\infty\}$ are binding conditions,
- $\vec{Y} := \{Y \to (\mathcal{T} \times \mathcal{C})^\infty\}$ are generation events ($\mathcal{C}$ is the set of all spatio-temporal components including $\perp$, a symbol for undefinedness), and
- $\vec{X} := \{X \to \mathcal{S}^\infty\}$ are exit conditions.

[8] The reason for the abbreviations Y and X is from the names entr**Y** and e**X**it.

The intuitive interpretation of B is that at each point in time all material *bound* (*i.e.*, whose parts are selected by the predicate) is connected to the given input, output, or mover. By definition (union set of I, O, M) only a single element can be bound by a binding. To bind multiple interface elements, multiple bindings have to be used (possibly with the same selection predicate). Examples of effects which can be modeled with these bindings include the following:

- For a conveyor belt all material sitting on top of it is moved at the same speed as the belt is running. The actual motion (stopping or specific speed) is modeled by a mover, the affected material (that is sitting on top) is bound by a selection predicate which includes all material that touches the surface, *i.e.*, collides with a thin volume on top of the belt.
- For a two part gripper, only material which is gripped should be moved. The gripping can be modeled by a selection predicate which only includes material touching *both* parts of the gripper, where touching again is modeled by collision with a thin volume.
- To model a gripper which operates by magnetism, the selection predicate has to be changed depending on the current power and enablement of the electric magnet.
- An RFID[9] reader can be described by a binding with a selection predicate selecting objects in the range of the reader. The corresponding material is then bound to a certain input of the reader component on which the data contained in the RFID tag is transmitted. As the RFID reader not only detects the presence of parts but also receives data, a detector can not be used for this case.

The details of binding as well as cases of material being bound by multiple bindings or a binding selecting multiple pieces of material is discussed later on.

Entries and exits (Y and X) are used to model environment assumptions. Each pair (τ, C') in a stream of $\vec{Y}$ indicates either the entry of the new component C' positioned by the transformation τ, or the lack of a new component indicated by $C' = \bot$. Similarly, each exit removes all material objects which are selected by its predicate. Both may depend on inputs and internal state of the component, so assumptions depending on the current state can be expressed. Examples for such assumptions may include that new material is only entered (into the machine) after an operator pressed a button (causing an input), or exits representing a combustion chamber might only work (*i.e.*, remove material) if the burner is on (*i.e.*, a certain internal state is reached). By using the empty predicate $\sigma : V \mapsto \boldsymbol{f}$, exits can be switched off temporarily.

The interface description given so far leaves many open questions, especially regarding the interplay of the extended component and the material. To explain the valid behaviors of a component C with semantic interface $F : \vec{I} \times \mathcal{A}(D) \to \mathcal{P}(\vec{O} \times \vec{D} \times \vec{P} \times \vec{M} \times \vec{B} \times \vec{Y} \times \vec{X})$ we derive a number of invariants on valid input/output pairs. For the remainder of this section let $(i, a) \in \vec{I} \times \mathcal{A}(D)$ and $(o, d, p, m, b, y, x) \in F(i, a)$. Relating this input and output is only allowed, if there is a stream $G \in (\mathcal{P}_{\text{fin}}(\mathcal{T} \times \mathcal{C}))^{\infty}$ such that the invariants given in the next paragraphs hold. G is used to

[9] Radio Frequency Identification

describe the set of (generated) material components at each time, with the pairs from $\mathcal{T} \times \mathcal{C}$ giving the position and component to be used. As material can only be introduced via the (finitely many) entries, at any point in time only a finite (although unbounded) set of components can be present in the system, hence the finite power set. In the remainder of this section we will be slightly loose in terms of notation to keep the formulas readable. For this we interpret the component C not only as the component's behavior function but also assume that it includes a concrete input and output stream and so fixes one possible execution of the component. Consequently, for a component C_G with $(\tau, C_G) \in G.t$ let I_G and O_G be the input and output channel sets of C_G, D_G the detector set, i_G and o_G denote the corresponding input and output valuations, and a_G and d_G denote the activations and positions of the detectors of C_G.

Remark 5.18
For a component C we denote by $ext_t(C)$ its *extent* which is defined as the union of the volumes of all its parts at time t. ⌟

The first two invariants characterize the components in the set $G.t$ at time $t \in \mathbb{N}$.

$$G.0 = \emptyset \tag{5.13}$$

$$\begin{aligned}(\tau, C_G) \in G.(t+1) \quad \Leftrightarrow \quad & \exists \tau' \in \mathcal{T} : \ ((\tau', C_G) \in G.t \ \vee \ \exists \psi \in y : \ \psi.t = (\tau', C_G)) \\ & \wedge \neg \exists \xi \in x : \ \xi.(t+1)(\tau(ext_{t+1}(C_G)))\end{aligned} \tag{5.14}$$

These invariants formalize the fact that new components may only be created via entries and removed by exits. Otherwise (first case) they have to be present already in the previous step, but possibly at another position.

The next invariant describes the movement of generated components from the set G for all $t \in \mathbb{N}$:

$$\begin{aligned}(\tau, C_G) \in G.t \quad \wedge \quad (\tau', C_G) \in G.(t+1) \quad \Longrightarrow \quad \tau = \tau' \quad \vee \\ \exists \beta \in B \, \exists \mu \in M \, \exists s \in \mathcal{S} : \ b(\beta).t = (s, \mu) \ \wedge \ s(\tau(ext_t(C_G))) \ \wedge \ \tau' = m(\mu).t \circ \tau\end{aligned} \tag{5.15}$$

This means, that the position of a generated component either does not change or may be changed according to a mover it is bound to. The mover may be any mover that is bound via a selection predicate colliding with the generated component's extent. In this case the position of the component is changed according to the transformation assigned to the mover in this step.

Similar to the position of generated components, their communication may be affected by bindings. For all $t \in \mathbb{N}$:

$$\begin{aligned}\forall (\tau, C_G) \in G.t \, \forall c \in I_G : \quad i_G(c).t = \perp \quad \vee \quad c \in O \wedge \\ \exists \beta \in B \, \exists s \in \mathcal{S} : \ b(\beta).t = (s, c) \ \wedge \ s(\tau(ext_t(C_G))) \ \wedge \ i_G(c).t = o(c).t\end{aligned} \tag{5.16}$$

$$\begin{aligned}\forall c \in I : \quad \text{neverbound}(c) \quad \vee \quad i(c).t = \perp \quad \vee \quad \exists (\tau, C_G) \in G.t : \ c \in O_G \wedge \\ \exists \beta \in B \, \exists s \in \mathcal{S} : \ b(\beta).t = (s, c) \ \wedge \ s(\tau(ext_t(C_G))) \ \wedge \ i(c).t = o_G(c).t \ ,\end{aligned} \tag{5.17}$$

where

$$\text{neverbound}(c) := \forall t \in \mathbb{N}\, \forall \beta \in B\, \forall s \in \mathcal{S} :\ b(\beta).t \neq (s, c)\ .$$

Here $\perp$ denotes an undefined input, which has to be contained in the carrier set of the corresponding type for the input. The equations state, that all inputs to the main component and the generated ones are either undefined or transmitted via a binding. The main component may exchange information with the generated components via channels if there is a binding for the channel and the generated component is currently bound, *i.e.*, its extent is selected by the selection predicate of the corresponding binding. The connection of channels then is based on the channel's identifier, *i.e.*, channels with the same name exchange information. The only major difference between both invariants is that for the main component there may also be inputs which are never bound to generated components, but instead receive inputs form the component's environment. These inputs are not restricted by the invariant.

The last three invariants correspond to Invariants 5.5 and 5.6.

$$\forall \delta \in D :\ a(\delta).t \ \Leftrightarrow\ ext_t(C) \bowtie d(\delta).t \ \vee \bigvee\nolimits_{(\tau, C_G) \in G.t} \tau(ext_t(C_G)) \bowtie d(\delta).t \tag{5.18}$$

$$\begin{aligned} &\forall (\tau, C_G) \in G.t\, \forall \delta_G \in D_G : \\ &\quad a_G(\delta).t \ \Leftrightarrow\ \Big(ext_t(C) \bowtie \tau(d_G(\delta).t) \ \vee \bigvee\nolimits_{(\tau', C'_G) \in G.t} \tau'(ext_t(C'_G)) \bowtie \tau(d_G(\delta).t)\Big) \end{aligned} \tag{5.19}$$

$$\begin{aligned} &\forall (\tau, C_G) \in G.t :\ \neg\, (ext_t(C) \bowtie \tau(ext_t(C_G))) \wedge \\ &\quad \forall (\tau', C'_G) \in (G.t \setminus (\tau, C_G)) :\ \neg\, (\tau(ext_t(C_G)) \bowtie \tau'(ext_t(C'_G))) \end{aligned} \tag{5.20}$$

Thus, all activations in any of the component must be caused by the parts of one of the other components and no parts of components may overlap. Different from Invariant 5.5 we use equivalence in Invariant 5.18, as we now assume the spatial part of the system to be not affected by any external influences (closed world assumption). Without this assumption, it becomes nearly impossible to make any conclusions about the material flow, as for example a human operator may change the material flow manually nearly without limitations.

Strong causality from Definition 5.1 easily carries over to this extended interface presented here, which allows us to define the extended component.

Definition 5.19 [Extended Spatio-Temporal Component]
Let (I, O, D, P, M, B, Y, X) be given as before and F a strongly causal function $F : \vec{I} \times \mathcal{A}(D) \to \mathcal{P}(\vec{O} \times \vec{D} \times \vec{P} \times \vec{M} \times \vec{B} \times \vec{Y} \times \vec{X})$. Then C described by the pair of both is called an *extended spatio-temporal component* or *material-aware spatio-temporal component*, iff Invariants 5.5 and 5.6 from Definition 5.2 hold and for all $(i, a) \in \vec{I} \times \mathcal{A}(D)$ and $(o, d, p, m, b, y, x) \in F(i, a)$ there is a stream $G \in \mathcal{P}_{\text{fin}}(\mathcal{T} \times \mathcal{C})^\infty$ such that Invariants 5.13 to 5.20 hold. ⌟

What is central in this definition is the stream G of generated components, which captures the material flow in the system. One possible interpretation is that this stream is determined by the environment semantics, which in turn abstracts physical laws. The definition allows a certain degree of freedom for the exact behavior enforced by the environment. For example Invariant 5.15

would also allow environments where motion of (generated) components is completely prevented. Similarly, Invariants 5.16 and 5.17 do not explain, which of the channels may provide a message if multiple components are bound to the same input channel. Our definition allows nondeterministic choice, but also environments with more complex priority rules.

Thus the extended spatio-temporal component introduces a family of models depending on the details of the modeled physics. The environment semantics may be chosen freely, as long as certain world assumptions (Invariants 5.13 to 5.20) are respected. One possible variant of the environment semantics is explained in more detail in Chapter 6. The next result confirms, that the extended spatio-temporal component is an *extension* of our basic model.

Lemma 5.20
Each spatio-temporal component can be interpreted as an extended spatio-temporal component by setting $B = Y = X = \emptyset$. ⌟

Proof. This is easily seen from Invariants 5.13 to 5.20 by setting $G = \emptyset^\infty$. □

5.3.2 Consequences of Dynamic Component Generation

This section discusses some of the consequences of introducing the extended spatio-temporal component. This includes the application of the composition operators for *normal* spatio-temporal components, an extension towards an even more general model, and a different view on systems based on the material flow.

Component Composition In the previous sections we introduced composition operators for spatio-temporal components, namely parallel composition (Section 5.2.2), data feedback (Section 5.2.3), positioning and mover connection (Section 5.2.4). Applying these operators to the extended spatio-temporal components is mostly straight-forward. For parallel composition we of course also require $B_1 \cap B_2 = \emptyset$, $Y_1 \cap Y_2 = \emptyset$, and $X_1 \cap X_2 = \emptyset$, *i.e.*, the syntactical interface of the composed components must be disjunct. Besides this, the merging of bindings, entries, and exits is performed just as with inputs or outputs by joining of the corresponding functions (*i.e.*, the projections of the resulting functions to the interface of one component yields its function, where function is one of the binding conditions, generation events, or exit conditions). The feedback operator only affects inputs and outputs and thus no changes are required. The only change in Definition 5.6 would be to complete the tuple (o, d, p, m) to $(o, d, p, m, b, y, x) \in \vec{O} \times \vec{D} \times \vec{P} \times \vec{M} \times \vec{B} \times \vec{Y} \times \vec{X}$. The situation is similar for positioning, but here we also have to respect the applied transformation for the selection predicates of the bindings and exits, and the initial transformations used with the entries. As the mover connection is built on positioning, the same definition for this operation can be used.

Types of Generated Components In the previous description the generated components were only of the non-extended type. This of course limits the model and certain effects can not be expressed. For example a multi-level material flow, where a brick (component) is on top of a pallet (component), would require the generated component (palette) to also provide bindings (for the bricks). Similarly, systems are imaginable where a generated component may spawn additional components. In both cases, also extended components have to be generated.

It is not too hard to extend the model to also allow the generation of extended components. However, the invariants are getting more complicated and some additional ones are required, as several new problems have to be accounted for. For example, multiple components may not form (directed) cycles in terms of binding each other. This would lead to unrealistic situations similar to the infamous stories of *Baron Münchhausen* who escaped from a swamp by pulling himself up by his own hair. Such cycles of course can not be created if only *simple* spatio-temporal components are generated.

Our compromise between expressiveness and complexity is to use semi-extended spatio-temporal components for the generated ones. These components support bindings, but do not have entries and exits. This way, only the cycle condition has to be respected in addition to the existing invariants. As the changes are minor we omit a formalization here. The operationalized model from Chapter 6 also implements this compromise.

Reduction of Component Interfaces to Material Flow The entries and exits introduce a different abstract view of components, where entries and exits are similar to input and output channels carrying material instead of data. The component's behavior then is reduced to material transformation. Such a perspective can lead to a theory of material transforming components, where composability also depends on spatial aspects, such as the relative positions of entries and *connected* exits, and the type and properties of material exchanged. Such a model could support assumption/guarantee reasoning in terms of material flow. The *assumption* predicate then defines expected material input at the entries, while the *guarantee* defines expected outcomes at the exits based on the assumption.

One drawback of such an approach is that only components of a certain complexity can be captured, *e.g.*, a conveyor belt would be similar to an n-element buffer, but a photo-electric barrier or a stopper can not be reasonably expressed in a material flow formalism. Another problem is the treatment of coordination components (like a supervisionary controller), which may affect composability. So, composability may depend on properties of additional components. For these reasons this thesis uses the rather simple material model introduced before, but ideas for more complex handling of material can be found in [BH10].

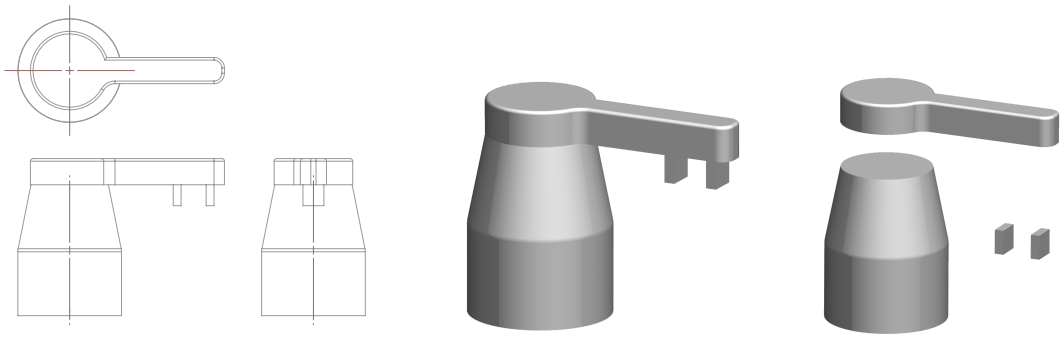


Figure 5.2: Several views of the robot: draft, 3D view, exploded view.

5.4 Examples

The presented formalism is not intended to be used to model systems, but rather as a means for reasoning about them. The modeling should be performed using some technique which is based on the semantics presented here. Nonetheless, we give an exemplary description of a system in this formalism to show how the presented concepts are applied. For all examples we assume the three-dimensional Euclidean space, which was described in Section 4.2.3.

5.4.1 Industrial Robot

Our first example is a simplified industrial robot, which is shown in Fig. 5.2. It consists of three parts: a base which can be rotated, the arm, which can be extended, and the gripper.

We model the base by a component C_{base} with syntactic interface $(\{i_{\text{base}}\}, \{o_{\text{base}}\}, \emptyset, \{p_{\text{base}}\}, \{m_{\text{base}}\})$. Via the input port, commands for rotation of the base are received, while on the output port it reports its current orientation in degree, thus $\text{car}(\text{type}(i_{\text{base}})) = \{\text{left}, \text{right}, \text{halt}\}$ and $\text{car}(\text{type}(o_{\text{base}})) = \mathbb{Z}$. The base behaves deterministically and is defined by $C_{\text{base}}(i) = \{(o, p, m)\}$, where o, p, m are constrained by ($t \in \mathbb{N}$)

$$o.0 = 0 \ ,$$

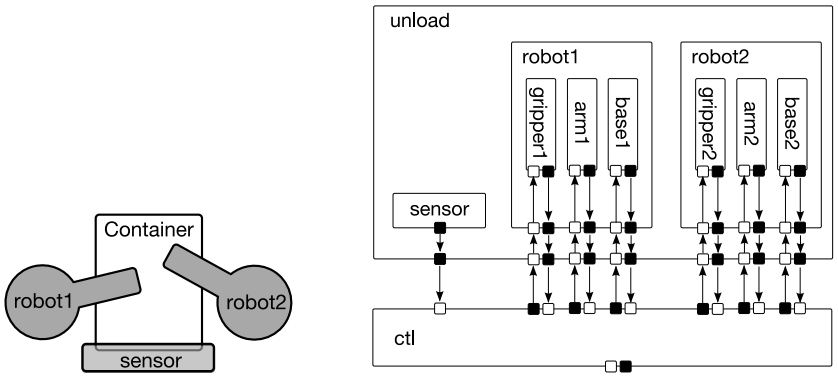
$$o.(t+1) = \begin{cases} \max\{(o.t) - 1, -90\} & \text{if } i.t = \text{left} \\ \min\{(o.t) + 1, 90\} & \text{if } i.t = \text{right} \\ o.t & \text{otherwise} \end{cases} ,$$

$p = p_0^\infty$, where p_0 is the volume corresponding to the base,

$$m.0 = \text{identity transformation} \ ,$$

$$m.(t+1) = \text{“rotation by } o.(t+1) - o.t \text{ degree”}.$$

So the base may rotate between -90 and 90 degree, but its shape is always the same. The transformations are chosen to always create the position which is reported by o.



(a) Graphical sketch of the setup as seen from the top.

(b) Component diagram for the two robots.

Figure 5.3: Setup for the *interacting robots* example.

The arm is described by the component C_{arm} with syntactic interface $(\{i_{\text{arm}}\}, \{o_{\text{arm}}\}, \emptyset, \{p_{\text{arm}}\}, \{m_{\text{arm}}\})$. Via the input port, commands for extending the arm are sent, while on the output port it reports its current length in centimeters, thus $\text{car}(\text{type}(i_{\text{arm}})) = \{\text{extend}, \text{retract}, \text{halt}\}$ and $\text{car}(\text{type}(o_{\text{arm}})) = \mathbb{N}$. The arm behaves deterministically and is defined by $C_{\text{arm}}(i) = \{(o, p, m)\}$, where o, p, m are constrained by $(t \in \mathbb{N})$

$$o.0 = 50 \ ,$$

$$o.(t+1) = \begin{cases} \min\{(o.t) + 1, 100\} & \text{if } i.t = \text{extend} \\ \max\{(o.t) - 1, 50\} & \text{if } i.t = \text{retract} \\ o.t & \text{otherwise} \end{cases} \ ,$$

$p.t$ is the corresponding volume with the arm extended to a length of $o.t$ centimeters,

$$m.0 = \text{identity transformation} \ ,$$

$$m.(t+1) = \text{“translation by } o.(t+1) - o.t \text{ centimeters”}.$$

The arm may be extended from 50 to 100 centimeters. Note that we do not model how this is achieved technically, but only how this affects the shape of the arm and the translation it applies to connected components.

The component C_{gripper} for the gripper is not given here, as it is similar to those given before. The entire robot can be expressed as

$$C_{\text{robot}} = (C_{\text{base}} \overset{m_{\text{base}}}{\circ\!\!-} C_{\text{arm}}) \overset{m_{\text{arm}}}{\circ\!\!-} C_{\text{gripper}} \ .$$

5.4.2 Interacting Robots

As a second example we model two robots that interact in unloading a container, which can be placed between them. A sketch is shown in Fig. 5.3(a) which also shows the sensor needed to detect the presence of a container. We use two components C_{robot1} and C_{robot2}, which are the same as the robot given in the previous example, but with the numbers 1 and 2 attached to all associated names[10]. Additionally we need a component C_{sensor} with syntactic interface $(\emptyset, \{o_{\text{sensor}}\}, \{d_{\text{sensor}}\}, \emptyset, \emptyset)$ characterized by $C_{\text{sensor}}(a) = \{(o, d)\}$,where $\text{car}(\text{type}(o_{\text{sensor}})) = \mathbb{B}$, $o.t = a.t$, and d is the stream giving the position of the sensor as shown in the figure. So the sensor is just used for converting spatial collision activations to logical data signals. The electro-mechanical part of the system is thus captured by

$$C_{\text{unload}} = C_{\text{sensor}} \parallel \text{pos}(C_{\text{robot1}}, \tau_{\text{robot1}}) \parallel \text{pos}(C_{\text{robot2}}, \tau_{\text{robot2}}) \ ,$$

where τ_{robot1} and τ_{robot2} describe the transformation used to place the robots to their intended position. In this setup however, there obviously is the possibility of the robots' arms to collide causing damage. For example for any input with $i_{\text{arm1}} = i_{\text{arm2}} = \text{extend}^{\infty}$ the result would be the empty set indicating an invalid state. One possible solution is the inclusion of a controller component C_{ctl}, which takes high-level commands and applies them to the robots, but at the same time tries to avoid collisions between them. It has the syntactic interface

$$(\{i_{\text{cmd}}, i_{\text{sensor}}, i_{\text{r1b}}, i_{\text{r2b}}, i_{\text{r1a}}, i_{\text{r2a}}\}, \{o_{\text{status}}, o_{\text{r1b}}, o_{\text{r2b}}, o_{\text{r1a}}, o_{\text{r2a}}\}, \emptyset, \emptyset, \emptyset) \ .$$

We define the partial connection relation q by the pairs

$$\{(i_{\text{r1b}}, o_{\text{base1}}), (i_{\text{r1a}}, o_{\text{arm1}}), \ldots, (i_{\text{base2}}, o_{\text{r2b}}), (i_{\text{arm2}}, o_{\text{r2a}}), (i_{\text{sensor}}, o_{\text{sensor}})\} \ .$$

A semantic interface for C_{ctl} is not provided here, but obviously the goal should be to define it in a way that the entire system $C_{\text{system}} = \text{fb}(C_{\text{ctl}} \| C_{\text{unload}}, q)$ specifies a total function, *i.e.*, there is no input leading to an invalid state (indicated by missing output).

As a summary the component compositions are graphically depicted in Fig. 5.3(b), which shows the hierarchical decomposition of the complex system to manageable components. The input and output ports of components are drawn as white and black little squares and the arrows indicate the data flow through the system. For the diagram we also added ports/channels for the grippers, which were not discussed before. This graphical representation also is the basis for the graphical syntax introduced in Chapter 6 and used in the tool implementation (*c.f.*, Chapter 7).

5.4.3 Adding Material

To conclude the examples, we add a belt conveyor, which is responsible for transporting the containers that are to be unloaded. The conveyor is modeled as an extended spatio-temporal component.

[10] Formally this could be captured by a renaming operation, however this kind of reuse should be handled by a modeling environment and thus we do not deal with it here.

We assume two selection predicates $s_{\text{top}}, s_{\text{end}} : V \to \mathbb{B}$ to be given which select components that are on the top of the belt or at its end. Both can for example be defined as collision selection predicates (*c.f.*, Definition 4.6) with corresponding volumes. The component C_{belt} has the syntactic interface $(\{i_{\text{run}}\}, \emptyset, \emptyset, \emptyset, \{m_{\text{belt}}\}, \{b_{\text{belt}}\}, \{y_{\text{belt}}\}, \{x_{\text{belt}}\})$. We do not model the rigid parts of the belt, but only its transportation logic here and thus the component C_{belt} has no parts. Additionally, we model only very simple physics here, so the belt may be switched on and off and can change its speed instantaneously. The decision whether the belt is running is performed via the input i_{run} which is of type Boolean, *i.e.*, $\text{car}(\text{type}(i_{\text{run}})) = \mathbb{B}$. The behavior of the belt for an input i is given as $C_{\text{belt}} = \{(m, b, y, x) \mid y \in Y'\}$, where $(t \in \mathbb{N})$

$$m.t = \begin{cases} \text{linear transformation in belt direction by } s \text{ centimeters} & \text{if } i.t \\ \text{identity transformation} & \text{if } \neg i.t \end{cases} ,$$

$$b = (s_{\text{top}}, m_{\text{belt}})^{\infty} ,$$

$$Y' = \{(c_i, t_i)_{i=0}^{\infty} \in (\mathcal{C} \times \mathcal{T})^{\infty} \mid \quad t_i = \tau_{\text{start}} \wedge (c_i = \perp \vee c_i = C^i_{\text{container}}) \wedge \neg\exists j, k \in \mathbb{N} : \; j \neq k \wedge |j - k| < 30 \wedge c_j \neq \perp \wedge c_k \neq \perp\} ,$$

$$x = s_{\text{end}}^{\infty} .$$

In this specification, the behavior of the mover is simple, as it applies a non-identity transformation exactly if the run signal is received. The speed of the belt is defined by the constant parameter s. Similarly, the binding with the selection predicate s_{top} to the mover is static, so every material which is on top of the belt will be affected by m_{belt}. The exit condition is static as well and will remove all containers that reach the end of the belt, recognized by s_{end}. The interesting part is the behavior definition of the entry, as here we use nondeterminism. It generates components from the family $C^i_{\text{container}}$ (for any $i \in \mathbb{N}$), which are components that model the containers. The initial transformation is τ_{start}, which we assume to be chosen such that the containers are positioned at the beginning of the belt. Each c_i can also be $\perp$ indicating that no new component is inserted. The last part (after $\neg\exists$) of the predicate ensures that no two containers are placed on the belt within 30 time steps, so there is a certain minimal generation delay.

As we can interpret the component C_{system} from the previous section also as an extended spatio-temporal component, the composition $C_{\text{system}} \| C_{\text{belt}}$ is feasible and results in a model of the unloading system with conveyor belt.

5.5 Discussion

To get a better understanding of the model presented in this chapter, this section discusses various aspects of the model. This includes the changes needed to switch to a continuous model of time, the degree to which our model captures reality faithfully, and the limitations of our modeling technique.

5.5.1 Continuous Time

We argued at the beginning of Section 5.2 that we explain our model using a discrete equidistant model of time. This time model simplifies some of the formulas as it allows for inductive definitions, which are hard (and often impossible) to provide with dense time. Using discrete time, however, can introduce deviations from the system which are not desired. For example, as message exchange only happens at time ticks, the exact timing is lost and errors up to the length of one time step can occur in the model. This can be problematic, *e.g.*, if the model is used to prove tight real-time boundaries on certain processing steps. In addition, the model respects collisions (between parts, or parts and detectors) only at discrete points in time. This can lead to situations, where due to fast motion an object is in front of a solid wall and at the next time step already behind it, *i.e.*, the part *tunnels* through the wall. Usually this should be avoided, but as there is no intermediate point in time where the object collides with the wall this is not detected. This fact violates spatial continuity for parts, *i.e.*, the observation that objects do not jump instantaneously from one position to the next[11].

In practice these problems with discrete time can usually be solved or neglected by choosing a sufficiently small time scale, *i.e.*, amount of time to pass between time steps. However, it is also possible to use a dense model of time, such as $\mathbb{R}^{\geq 0}$, to eliminate the aforementioned problems. While FOCUS can be easily carried over to dense streams, in our modeling theory more effort is required for some of the definitions. For the basic model of spatio-temporal components, all of the definitions (including the invariants from Definition 5.2 and the composition operators) can be used with dense time as well, only that now $t \in \mathbb{R}^{\geq 0}$ instead of $t \in \mathbb{N}$. The only difference is for the positioning operator and the movers, which use streams of differential motion for each time step. In the discrete time model we can use a transition function for each discrete time step. For continuous time, however, at a single point in time the transformation is essentially 0 (*i.e.*, identity transformation). Instead, each mover would be assigned a stream of transformation derivatives[12]. For this we would have to extend our model of a transformable collision space by also requiring the set of transformations to be differentiable, as well as an additional set of transformation differentials and an integration operator (required for the integrated transition stream from Definition 5.8).

For the extended spatio-temporal component there is a second challenge related to the generation of material and material flow. First, we want the streams of generation events assigned to each entry to be *sparse*, *i.e.*, a component (non-$\perp$ value) may only occur a finite number of times in each finite interval of time. This ensures that at any point in time only a finite number of components is present. Consequently, the stream G will also be based on continuous time, which requires the invariants describing changes in G over time (namely 5.14 and 5.15) to be modified to using a differential representation. Of course the motion of material (Invariant 5.15) also needs to be adjusted to the differential transformations from the extended space model.

[11] This is at least true at the level of classical mechanic. Different laws might apply if such a model would be used to describe effects at a subatomic level.

[12] This is also the point, where we need *continuous* time and not just *dense* time. See Section 4.3.1 for the difference.

While the described changes can lift the model to continuous time, the practical gain is rather low. At least for typical simulation applications, the evaluation of the model happens again step-wise as the equations involved typically do not allow for an analytical solution but rather require numerical methods. Also the algorithms typically used for collision detection can only handle static scenes at a single point in time[13] and thus are applied at discrete time steps.

When using discrete time, one important factor that should be respected is to describe all functions involved independently from the step size used (or rather treat the step size as a parameter). This way the step size can easily be decreased in case of simulation artifacts and even adaptive simulation methods can be applied, which sample using a smaller step size in regions with fine-grained interaction. Also for closed analytical methods, often the additional complexity introduced from a continuous time model does not justify the often minor gain in terms of precision.

5.5.2 Adherence to Reality

An obvious question when providing a new description technique for real-world systems is to what extent the created models correspond to the real systems. Of course this heavily depends on the skills and intent of the modeler, as bad models can be created with any modeling approach. Thus, we have to limit this discussion to the concepts provided by the modeling theory. As certain aspects of the theory are adopted from FOCUS, we will limit this discussion to our extensions and modifications.

The selection of the primitive interface elements of parts, detectors, and movers (besides inputs and outputs) corresponds to the spatial effects we consider the most relevant, namely spatial extent, spatial presence, and motion. These elements can also be mapped to those of electro-mechanical systems, which are rigid structure, sensors, and actuators. As all of these are externally observable, they are modeled as part of the component interface. On the level of these interfaces our invariants capture the facts that parts (solid objects) can not overlap in space, and that detectors work faithfully (*i.e.*, they will report the presence of parts). Both of these are consistent with real-world observations. The composition operators are constructed to keep as much of the behavior of the composed components, while respecting these invariants.

The additional elements from the extended model seem to be slightly more artificial. Especially the entries and exits are not an actual part of a system. Instead, these elements are used to capture assumptions and guarantees of the system's environment. While in practice often these material flow constraints are not explicitly formulated, of course every system does only work correctly in the presence of certain kinds of material flows (it is usually unspecified what happens when bricks are inserted into a bottling plant). The additional invariants ensure the consistency of the material flow between entries, exits, and the system's model in terms of continuity (material does not just (dis)appear within the model). Finally, the bindings are used to make interaction points with material explicit. Often it is just taken for granted that certain events in the system (such as a

[13] While there are algorithms that respect the motion over time, these typically only work for very limited sets of shapes and/or transformations.

belt conveyor running) will have effects on the material. But these effects are caused by physical interactions, such as friction on the belt or a light ray being reflected. While many simulation systems attempt to incorporate these effects into their simulation semantics, we chose to make them explicit in terms of bindings. As our models are targeted to the design phase of system's development, we consider it important that these effects are intentionally chosen and thus modeled and documented explicitly.

While the existing invariants disallow certain interactions that are not consistent with real-world observations, many other impossible situations can be modeled[14]. Examples of this are violations of gravity, objects being moved without any visible physical device (only using an "invisible" mover), or impossibly fast acceleration of parts. There are two main reasons for not adding more restriction to disallow this behavior, which are both rooted in the fact that our model is intended to be a specification language rather than an implementation language. One reason is that as more physical laws are enforced, the more detail is required. As soon as forces and momentum are included, also the mass of objects is needed, while the inclusion of friction will require friction coefficients. Often these details are not easily available at an early design stage, and even if they were, these details could distract the designer from the main task of describing the functional view of the system (and not the realization). The second reason is that usually the system is not designed in a big-bang manner, but rather described incrementally. The violation of physical laws allows to replace parts of the system by extremely abstract models. For example, to transport material from one processing station to another, a simple model could just let the material fly (via a mover) to its destination. Later, this model could be refined to use conveyors or robots for transportation, but until then this simple model helps to bridge the gap between the (already modeled) stations.

5.5.3 Limitations

The previous section described constructs in our model which go beyond the possibilities of real-world systems. To complement this view, we discuss real-world observations which are impossible or at least hard to capture in our approach. The most obvious ones are physical effects from areas we did not consider in the first place. Examples are temperature and heat exchange, or electromagnetic waves and effects. For these effects our model is as good or bad as behavior models for plain software systems, as we have to somehow encode them in the state space of the model, leading to the usual problems, such as weak composability. In some cases the spatial aspects of our model might even help, *e.g.*, heat exchange is expected to mainly happen between parts which are next to each other in space. We discuss possible extensions in this direction in the outlook in Section 9.4.

Another group of properties is at least related to space. These are velocity (first derivative of position), acceleration (second derivative of position), and force (which depends on acceleration and

[14] This problem is not specific to our model and is also known from existing modeling techniques for software systems. For example, it is easy to specify timing constraints which can not be met with any implementation. Similarly, FOCUS allows the specification of systems that solve undecidable problems.

mass), which are not primary elements in our model. While velocity and acceleration can be expressed by the rate of change in the transformations applied at each time step, force is harder to express, especially in combination with preservation of momentum and energy. One reason to not deal with force explicitly was described in the previous section, *i.e.*, its inclusion usually also involves mass and often friction, which leads to many additional details required in the models. The second reason is that at a functional level often the exact force or momentum is not required, but only the resulting effect (or function) on a logical level is relevant. For example it is sufficient to known that a stopper will stop queued material from further movement. The exact force the material exercises on the stopper is only relevant if we are interested whether the stopper will be deformed by this force. But this question is typically not answered by a functional behavior model, but rather a specific (and purely mechanic) model, such as models for the finite element method.

There are also purely spatial properties, which we can not express. The reason for this is that our spatial properties are based on collision of volumes (subsets of space) and possibly Boolean combinations of these. A prominent example of a component that can not be modeled this way is a distance sensor, as our model does not support measurement of distance. What could be modeled, is a sensor detecting whether the distance is above or below a certain threshold, as this is a Boolean property. This would be expressed by a detection of exactly the desired length and a collision with this detector (or the absence of a collision). But if there is a complex functional dependency between the distance and some controlled property in the system, this can not be directly expressed.

Another source of limitation is rooted in our material model which is focused on discrete rigid objects. Our approach will make it complicated to model liquids, loose goods (such as sand), or abrasion processes in machine tools. All of these can be approximated to some extent, for example by treating a liquid as if it consisted of unit sized chunks, but this simplification of course will obfuscate some of the specific properties of these kinds of material. Interestingly, these kinds of material are also very rare in pure material simulation systems, which concentrate on the specifics of material without dealing with the behavior of the overall machine. Furthermore, the majority of systems in the automation domain deal with discrete rigid objects and thus our approach is widely applicable, even if not every kind of machine can be modeled with it.

5.6 Summary

This chapter described our modeling theory for space-intensive mechatronic systems, which is the core contribution of this thesis. The model is a spatio-temporal extension of the stream-based formalism FOCUS and provides a black-box view of the system, as it only relates externally visible elements and events. As such it is a suitable candidate for modeling systems at a functional level, where not the implementation but the specification of functions is the primary concern. Our model addresses both spatial properties and material flow, two aspects which are fundamental for modeling systems in the automation domain.

As already pointed out in Section 5.4, the technique described in this section is not useful for direct application to system modeling. Especially the geometric properties are not easily expressed in mathematical notation, while drawing a picture of the geometric configurations is often simple. Instead, the model presented here is intended as the semantic foundation of an operationalized model which can be supported by a modeling tool. One possible operationalization of the modeling theory is presented in the next chapter, which is also used as the basis for a prototypical tool in Chapter 7.

This chapter also discussed various deviations from the real world and limitations in expressiveness. The impact of these limitations on the actual modeling process is hard to judge from a purely theoretical perspective, as the exclusion of certain aspects (by abstraction) is one of the goals of modeling. Furthermore, it is not clear how often the *problematic* constructs occur in practice and how hard they are to circumvent. A partial answer to these questions can be obtained by applying the modeling theory (or its operationalization) to real systems to see whether and how easy (or complicated) they can be modeled in our formalism. This was done in several case studies, which are described in Chapter 8.

6 An Operationalized Model

While the theory introduced in Chapter 5 is sufficient for describing models of mechatronic systems and also clearly defines the semantics of these models, for practical modeling this mathematical style of modeling is often cumbersome and hard to understand, especially for non-mathematicians. Additionally, the richness of the underlying language of mathematics makes the implementation of the theory in an engineering tool nearly impossible.

To bridge this gap, this chapter describes an operationalization of the modeling theory by providing a concrete meta-model (*i.e.*, a description of the valid modeling elements and their allowed structural relationships) as well as the semantic meaning of this meta-model in terms of the modeling theory explained before. By limiting the modeling language to certain meta-model elements, it becomes easier to apply as the possible choices for a user are reduced. This also is the main step towards tool support, as a modeling tool can be limited to the meta-model elements, instead of the full body of mathematics. When defining the meta-model it must be ensured that the expressiveness of the language is not reduced too much, so it can still capture the relevant properties of real-world systems. The validation of sufficient expressiveness is mostly provided by Chapter 8 which applies the tool described in Chapter 7 and thus this meta-model to real-world systems modeling. It is also worth noting, that certain limitations, such as the use of static shapes or the limitation of motion to certain axes, which are described later on, move some modeling options from the dynamic behavior view (called semantic interface before) to the static aspects of the component (syntactic interface). This chapter in conjunction with Chapter 5 forms the core contribution of this thesis.

The following sections provide an overview of the meta-model and explain the choices that were taken where multiple design options were available. The meta-model is based on the one of AutoFOCUS [BHS99, SPHP02], which is a tool implementation of the FOCUS theory (*c.f.*, Section 5.1). Where appropriate, the discussed parts of the meta-model are complemented by UML class diagrams. These diagrams do not provide all details to keep them easier to understand. Missing details and especially constraints are explained in the text, as formal constraints (*e.g.*, formulated in OCL[1]) are often hard to understand. The model is sometimes illustrated by screenshots taken from its tool implementation, thus anticipating parts of Chapter 7. These images also suggest a possible graphical syntax that can be used with this meta-model. The chapter is concluded by an example which is explained in full detail. It may be helpful while reading this chapter to refer to this example from time to time. Further examples can also be found in Chapter 8.

[1] Object Constraint Language

6.1 Types, Space and Time

While the previous chapters discussed various options for modeling time and space and dealt with types by means of rather abstract carrier sets, we provide more details here. Of course the variant used here is only one of a plethora of options.

6.1.1 Types and Type System

The area of types and type systems has been studied in great detail in the last decades (for an introduction and overview see, *e.g.*, [Pie02]). As the focus of this thesis is on the interplay of spatio-temporal properties and behavior models, the details of the type system used are of minor concern. Thus a simple and clean functional style type system has been used, which is nonetheless capable of expressing enumeration types, record types (sometimes also referred to as tuple types), and recursive data types, which can be used to describe lists or tree-like data structures.

The type system used is based on the one used in the current AutoFOCUS3 tool prototype, which itself is a simplification of the type system used in the first AutoFOCUS implementation. Thus, the type system itself is not a contribution of this thesis, but will be recapitulated shortly as the following sections are partially based on it.

Types A type is specified by its name and a set of constructors, which consist of a name and a list of types each. The entries of this list are called the constructor's *parameters*. To define the carrier sets of the types we assume that the types $T = \{t_1, \ldots t_n\}$ are given and t_i has constructors $c_i^1, \ldots c_i^{m_i}$. We denote by $|c_i^j|$ the number of parameters of constructor c_i^j and by $\text{type}_i^j(l) \in T$ the type of its l-th parameter. Then we define for each $k \in \mathbb{N}$ the mapping car_k from types to their carrier set approximation as follows. The first function car_0 maps the type t_i to the set containing all its parameter-less constructors (*i.e.*, those with an empty type list). For $k > 0$ the function car_k is inductively defined as

$$\text{car}_k(t_i) := \text{car}_{k-1}(t_i) \cup \bigcup_{j=1}^{m_i} \left\{ c_i^j(p_1, \ldots, p_{|c_i^j|}) \mid \forall l \in \{1, \ldots |c_i^j|\} : \; p_l \in \text{car}_{k-1}(\text{type}_i^j(l)) \right\} \quad .$$

So car_k maps each type to all type correct terms which can be constructed from its constructors using only terms from car_{k-1} as parameters. The carrier set for the type t_i is then given by $\bigcup_{j \in \mathbb{N}} \text{car}_j(t_i)$, *i.e.*, by the limit of the finite approximations of the type.

As notational convention we write a constructor by giving its name followed by the parameter list (list of types) enclosed in parentheses. If a constructor has no parameters, the empty parentheses may be omitted. A type is written by its name followed by an equal sign and a list of constructors separated by vertical bars |. The list is concluded by a semicolon. Typically, the names of types start with an upper case letter, while constructors start with a lower case letter. We complement the formal type definition by some examples of types and their carrier sets.

Example 6.1
The simplest types to be defined are those having only empty constructors, also known as *enumeration types*.

```
Color = red | green | blue ;
Direction = north | east | south | west ;
```

The carrier set of *color* is then defined as $\{\text{red}, \text{green}, \text{blue}\}$, while the one of *direction* consists of $\{\text{north}, \text{east}, \text{south}, \text{west}\}$. Types which are described by constructors that do not lead to recursion, *i.e.*, no constructor has a parameter of the type itself or a type containing a constructor referring to the type, are usually called records or *tuple types*:

```
DirectionTriple   = dirTriple (Direction, Direction, Direction) ;
ComplexDirection = noDirection | simpleDirection (Direction) |
                   colorDirection (DirectionTriple, Color) ;
```

The carrier set of *DirectionTriple* contains besides others the elements dirTriple (North, North, West) or dirTriple (South, South, South). The carrier set of *ComplexDirection* contains elements noDirection, simpleDirection (East), and colorDirection (dirTriple (North, North, West), red). Types like *ComplexDirection* which combine multiple record-like constructors are sometimes referred to as *variant types* or *union types*. Without limiting recursion, the type system can also describe lists or tree-like structures.

```
ColorList = emptyColorList | colList (Color, ColorList) ;
Tree      = emptyTree | consTree (Tree, ColorList, Tree) ;
```

While the previous examples of types all had finite carrier sets, the carrier sets of *ColorList* and *Tree* are both infinite (but countable). For the *ColorList* type emptyColorList, colList (red, emptyColorList), or colList (blue, colList (green, emptyColorList) are examples of elements. It is also possible to define types with an empty carrier set:

```
Empty1 = ;
Empty2 = infLoop (Empty2) ;
```

While *Empty1* is trivially an empty type, *Empty2* is empty because none of the finite approximations of the carrier set contains any elements as no empty constructor exists. While the infinite term infLoop(infLoop (...)) would be a fixed point of the type equation, it is excluded from the carrier set by the inductive definition used. ⌟

Remark 6.2
We expect the primitive types *boolean* (true or false), *int* (integer numbers), and *double* (real numbers) to be defined. The type *boolean* is given by **boolean = true | false**, and the types *int* and *double* by implicit lists of a countably infinite number of constructors resembling the sets $\mathbb{Z}$, respectively the set of all rational numbers $\mathbb{Q}$ written in their decimal expansion. Additionally, for each type the equivalence test $==$ is available and yields a *boolean* result. ⌟

Functions While the notion of a type given so far is sufficient to characterize the carrier sets and thus fills the gap left in Chapter 5, we also need ways to operate on types. This is performed by functions[2], which describe rules for transforming a list of terms into another term. A function is described by a name, a parameter list (just as with constructors this is a list of types), and a list of transformation rules. Each transformation rule is given by a list of *match terms* and a *result term*. The match terms may be constructed only from constructors and variables, while the result term may be constructed from constructors, functions, and the variables used in the match term. The *input* of a function consists of a list of elements from the carrier sets of the corresponding types of the parameter list. A transformation rule is said to match an input if there is a valuation for the variables in its match terms such that each match term is the same as the corresponding input value. The result of a function for a given input is the result term of the first matching transformation rule, where each occurrence of a variable in the result term is replaced by the valuation used during the matching step.

It is easily seen, that by using these evaluation rules together with the carrier set construction described before, the result of a function for a single input is uniquely determined, if at least one transformation rule matches. We only allow *total* functions, *i.e.*, each function must have a matching transformation rule that terminates after a finite number of steps for every syntactically valid input.

For notation, functions are declared using their name followed by the parameter list in parentheses, a colon, and the result type. The transformation rules are separated by vertical bars | from each other and the preceding elements, and the match and result terms are separated by $->$. The function declaration is concluded by a semicolon. Variable names consist of lower case characters and start with the underscore, function names start with a lower case character.

Example 6.3
We build upon the types declared in Example 6.1. A function for giving the next clockwise direction would look as follows:

```
clockwise (Direction) : Direction
  | north -> east
  | east  -> south
  | south -> west
  | west  -> north ;

opposite (Direction) : Direction
  | _d    -> clockwise (clockwise (_d)) ;
```

The second function applies the *clockwise* function twice. An example using more match terms would be an if-then-else function, which returns the second or third parameter based on the value of the first boolean parameter.

[2] In the context of the type system we use the term *function* not only in the strictly mathematical meaning, but also to denote an element of the type system's meta-model. Of course these type system functions can be interpreted as mathematical functions, but it is important to differentiate them at least during their introduction.

```
ite (boolean, ColorList, ColorList) : ColorList
  | true,  _a, _b -> _a
  | false, _a, _b -> _b ;
```

By using recursive functions, also more complex problems can be solved, especially when dealing with recursive data structures. The following example removes the first occurrence of a color from a list.

```
delColor (ColorList, Color) : ColorList
  | emptyColorList, _c       -> emptyColorList
  | colList (_c1, _l), _c2 ->
        ite (_c1 == _c2, _l, colList (_c1, delColor (_l, _c2))) ;
```

⌟

For the primitive types the most common operations (boolean connectives and basic arithmetic operations and comparisons) are defined. Instead of the functional notation, we use the operator (infix) notation in these cases. For these operations the syntax and operator precedence known from C and Java is used. As the exact details are not relevant here and the operators used in examples are mostly self-explanatory, they are not discussed in more detail.

6.1.2 Space

While in the previous chapter the rather abstract notion of a transformable collision space was used, which is fulfilled by many models of space, here we fix a concrete space. Similar to the type system, the goal is to have a specific model of space which is simple but sufficiently expressive so we can study the spatio-temporal behavior model and its expressiveness without having to deal with a too complex space model. For practical applications the model of space can be chosen more complex without changing the entire behavior model too much, however, as Section 8 demonstrates, most systems can be easily expressed using the space model outlined in the following paragraphs.

The model is based on the well-known three-dimensional Euclidean space $\mathbb{R}^3$. Let $\mathcal{T}_t := \{f : \mathbb{R}^3 \to \mathbb{R}^3 \mid \exists v \in \mathbb{R}^3 : \ f : x \mapsto x + v\}$ be the set of translations and $\mathcal{T}_r := \{f : \mathbb{R}^3 \to \mathbb{R}^3 \mid \exists M \in \mathbb{R}^{3\times3} : \ M^T = M^{-1} \wedge \det(M) = 1 \wedge f : x \mapsto Mx\}$ the set of rotations[3] in the three-dimensional Euklidean space. We define the set of all transformations used by $\mathcal{T} := \{f \circ g \mid f \in \mathcal{T}_r, g \in \mathcal{T}_t\}$, *i.e.*, any combination of translations followed by rotations.

Remark 6.4
All of $(\mathcal{T}_t, \circ)$, $(\mathcal{T}_r, \circ)$, and $(\mathcal{T}, \circ)$ are groups. Furthermore, the sets $\{f \circ g \mid f \in \mathcal{T}_r, g \in \mathcal{T}_t\}$ and $\{g \circ f \mid f \in \mathcal{T}_r, g \in \mathcal{T}_t\}$ are equal, so it does not matter if we apply the translation first or second. The possible transformations in $\mathcal{T}$ are the same in both cases. ⌟

[3]The property $M^T = M^{-1}$ ensures orthogonality of the matrix and $\det(M) = 1$ makes it a rotation matrix. An equivalent definition of rotations can also be obtained using *unit quaternions* (*c.f.*, [Alt05]).

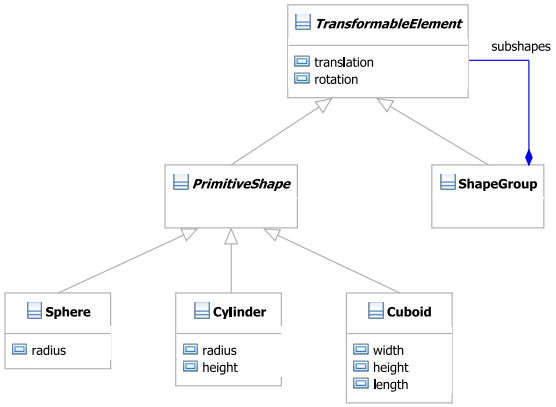


Figure 6.1: Meta-model for volumes definition.

For the volumes we assume a set P of primitive volumes, which contains besides others the spheres $S_r := \{x \in \mathbb{R}^3 \mid |x| \leq r\}$, the cylinders $C_{r,h} := \{(x,y,z)^T \in \mathbb{R}^3 \mid |(x,0,z)^T| \leq r \wedge |y| \leq h\}$, and the cuboids $B_{w,h,l} := \{(x,y,z)^T \in \mathbb{R}^3 \mid |x| \leq w \wedge |y| \leq h \wedge |z| \leq l\}$. This set can be extended by other compact and fully dimensional subsets of $\mathbb{R}^3$, which is used in Chapter 7 when importing geometry from CAD data. We define the set V of volumes as $\mathcal{P}_{\text{fin}}(\{t(p) \mid t \in \mathcal{T} \wedge p \in P\})$, *i.e.*, finite sets of transformed primitives. The empty volume V_0 is represented by the empty set. The collision relation is defined for $v_1, v_2 \in V$ by

$$v_1 \bowtie v_2 \;:\Leftrightarrow\; \exists x_1 \in v_1, x_2 \in v_2 : \dim(x_1 \cap x_2) = 3 \;,$$

i.e., there is an actual penetration of the objects resulting in a fully dimensional intersection, not a mere touching of volumes. Furthermore, we define volume union as set union, and the set of transformations $\mathcal{T}$ is just extended point-wise to the elements of V. Checking all items of Definition 4.1 easily confirms the following result, which ensures that the space definition given so far is compatible with the theory from the previous chapters.

Lemma 6.5
The tuple $(V, 0_V, \bowtie, \sqcup, \mathcal{T})$ *give before is a transformable collision space.* ⌟

The actual meta-model shown in Figure 6.1 resembles the mathematical model of volumes described before. The only deviation is the hierarchical organization using the *ShapeGroup* (composite pattern), but as the transformations form a group we can combine all transformations along the paths in the hierarchy and thus the hierarchical representation is equivalent to the volumes V. The reason for this choice of space is to simplify the transition to a tool implementation, as the shape model closely resembles commonly used scene graphs and the collision test corresponds to the one usually implemented in existing libraries for computational geometry.

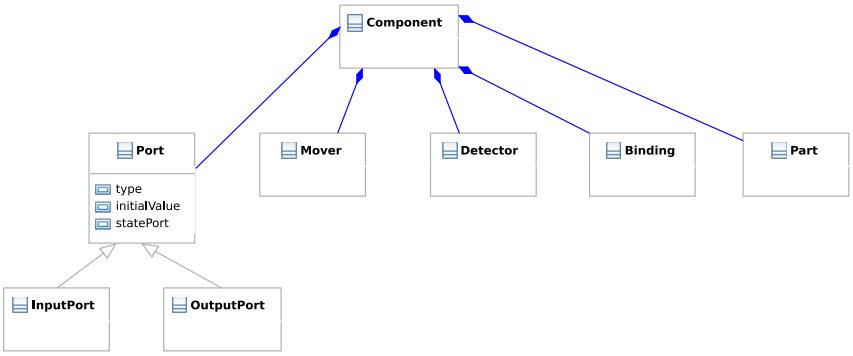


Figure 6.2: Meta-model for syntactic component interface definition.

6.1.3 Time

Regarding time, there is not much to add as we already decided in Section 5.2 on the use of a linear discrete time. However, we want to revisit the discussion of Section 5.1.3 regarding the use of strong versus weak causality. There, we decided to stick with strong causality as it simplifies the theoretic framework and especially the composition of components. But, as discussed at the same place, strong causality introduces the problem of delay accumulation, which means that a chain of n components slows down the propagation of a signal by at least n time ticks. This can lead to unexpected problems in actual modeling, usually when long chains of simple components are used to realize a complex signal processing functionality. For example, two signals sent at the same time at different ports of the same component may reach another component at two different times depending on the number of components on each signal path.

One tool to remedy this situation is the controlled inclusion of weakly causal components into the meta-model. Thus, a component may be declared as weakly causal, which causes its outputs to depend also on the inputs from the same (instead of the previous) time step. To ensure composability, we require that no (directed) cycles of weakly causal components may exist, *i.e.*, each cycle must include at least one strongly causal component. It can easily be checked that composition is still well defined as the outputs and thus inputs for all components can still be calculated inductively from the previous time step. However, now the order in which the components are evaluated does matter for the weakly causal ones.

6.2 Static Aspects

The core element of the model is the *component*. A component can be used to describe individual parts of the system as well as the entire system. The main elements of a component's *syntactic interface*, which describe its overall structure, are shown in Figure 6.2 and follow the definition of

an (extended) spatio-temporal component from Chapter 5. The *ports* describe the communication endpoints of the component, where communication includes signal exchange between controllers via a common bus as well as the transmission of the number of revolutions from a motor to the belt conveyor it is driving. The ports are annotated by a type giving the kind of messages exchanged, as well as an initial value used for the first simulation tick. The spatial extension of the component is given by *parts*, which describe the dimensions and shape of the component. These are the portions of the system traditionally modeled in 3D-CAD tools, although we usually use simplified geometry for our abstract models. Locations in space where the component can observe and react to the presence of other parts are marked by *detectors*, and where material is affected by movers or for data exchange are described by *bindings* (*c.f.*, Section 5.3.1) which are detailed in 6.5.2. The remaining elements, movers, denote facilities which affect the position of other parts or material.

6.2.1 Signal and State Ports

In pure software systems usually an *event based* principle of communication is used, *i.e.*, messages are only transmitted in response of certain events or state changes and thus often no signals are sent at all. The opposite is the *state based* principle where the current state is continuously reported. This is commonly found in simple sensors but is also used to model mechanical transmission, *e.g.*, the speed torque of a motor which is *sent* (as we model this by signal exchange) to a connected gear box, where the absence of a *message* is not sensible.

As we represent both software and hardware in the model, both styles of communication are supported. Whether event or state based communication is used is stored at the port in the boolean *statePort* attribute. As long as only ports of the same communication principle are connected, this flag only indicates whether the empty message ϵ, which indicates the lack of information, may also be sent or received via this port. However, it is also possible to connect ports using different communication styles. If the sender is state based, this is no problem, as the receiver should also work if no ϵ messages are received. In the other case, an event port sending to a state port, the ϵ messages have to be suppressed, as the state port does not accept them. This is performed by buffering the last non-ϵ value and reusing it in the case of an ϵ message.

6.2.2 Geometry Integration

One goal of our model is the integration of space and behavior. The link are the parts, detectors, and bindings which belong to a component and are assigned volumes of our space system. Figure 6.3 shows how the model elements from Figures 6.1 and 6.2 are integrated in the meta-model. By inheriting from *TransformableElement*, a component can have a local transformation attached. The parts, detectors, and bindings are just made containers for multiple volumes by inheriting from *ShapeGroup*. They may be interpreted as individual volumes in our space system by application of the volume union operator.

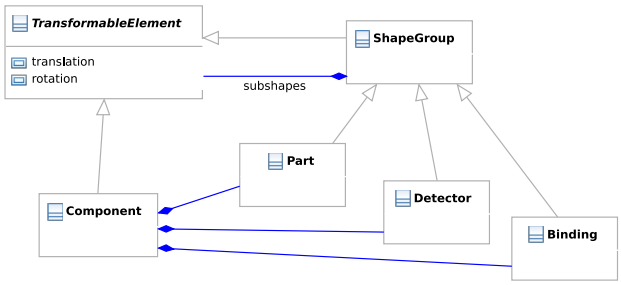


Figure 6.3: Meta-model for integration of geometry with components.

The main difference to the model presented in Chapter 5 is that parts, detectors, and bindings maintain a fixed shape but may be translated and rotated over time, while the mathematical model just maps these elements to an arbitrary volume in each time step, which can also express transformations but also more complex shape changes. There are two reasons for this limitation. First, the systems from the domain of factory automation being our primary example consist of rigid incompressible system elements, so a change in shape is rarely needed. Second, the description of a continuous change in shape requires a model of space supporting complex parametric geometry descriptions, which are seldom found even in commercial construction systems. Thus, to keep the meta-model of manageable size and the tool implementation within scope, the static shape model was chosen.

The limitation to static shapes is also the reason why the geometric aspects are explained in the section on the syntactic interface. Of course there are cases even for factory automation systems, where the shape of parts changes. One is the deformation of parts in case of collisions as a consequence of system malfunctions, the other is the change of the workpiece based on processing steps, such as drilling or grinding. The first use case is seldom analyzed in factory automation, but more of a concern in automotive systems with their extensive crash tests. The second application of changeable shapes is more relevant to the domain. One solution would be to allow the specification of multiple shapes for each part, detector, and binding and switch between them based on the behavior of the component. A more versatile solution would be to integrate existing solutions from finite elements or chipping and abrasion simulation. While both are clearly interesting directions for future work, they were not pursued in this thesis.

6.2.3 Movers and Axes

Similar to the fixed geometric shapes used for parts, the transformations used by movers are limited to a certain set described by an axis. In Figure 6.4 this one to one relationship between movers and axes is emphasized. The actual details, which parts are affected by a mover and its axis, and

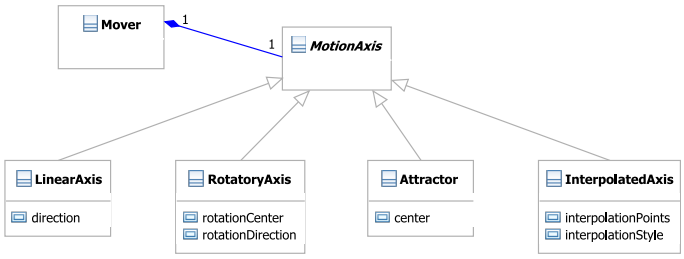


Figure 6.4: Meta-model for mover definition via axes.

how much the position of a part is transformed by a mover are explained in Section 6.3.1. Here we concentrate on the static aspect, which consists of the definition of possible transformations.

There are two basic uniform transformations, which are independent of the element the mover is applied to: linear motion (translation) and rotatory motion. These two, which correspond to the sets $\mathcal{T}_t$ and $\mathcal{T}_r$ from Section 6.1.2, are represented by the *LinearAxis* and the *RotatoryAxis* in the meta-model. In theory, each valid motion can be composed from these two. However, certain motion paths are complex to describe by a combination of rotations and translations only. Examples include the motion of material on top of a curved conveyor or of parts connected to a belt attached to multiple pulleys (both cases were actually encountered in the models discussed in Chapter 8). To simplify the modeling of such cases, the definition of a motion path using interpolation points and a interpolation scheme (*e.g.*, B-spline interpolation) is supported via the *InterpolatedAxis*. The transformation of an element then depends on its relative position to the path and the transformation is calculated to move the element forward along this path. More concrete, for a moved element the nearest point on the path is determined and the transformation is calculated based on the tangent of the curve on that point. The axis has further attributes (not shown in the diagram) to control whether the calculated transformation should be linear or rotatory. In the first case the orientation of an element will be kept, while using rotatory transformations will make the elements orientation follow the path. Another example of an axis whose transformation depends on the position of the moved element is the *Attractor*, which translates an element in the direction of the Attractor's center. By negating the direction of motion, this can also be used as a repeller. Such an axis could for example be used when modeling an electromagnet, which can be switched on and off.

The four axis types described so far were sufficient for modeling all examples of real-world systems observed in Chapter 8. However, it is possible that certain effects in a system need another type of motion. The model supports possible extension in this direction, as only another subclass of *MotionAxis* has to be provided.

comp StopperUnit	
inport:	mode: EMode
inport:	request : **boolean**
outport:	close_stopper : **boolean**
outport:	part_present : **boolean**
part:	stopper_part
detector:	_sense1
binding:	_stop_binding
mover:	_move_stopper: linear axis

Figure 6.5: Syntactical component specification

6.2.4 Textual and Graphical Syntax

The syntax used for the definition of the component's syntactic interface is demonstrated in the example shown in Figure 6.5. The component is drawn as a box, with the ports shown as small circles on its border. Where necessary, the ports can also be labelled to make them distinguishable. More information is contained in the table on the right, which lists all syntactic elements of the component together with their names. The geometry for the part, the detector, and the binding as well as the details of the linear axis of the mover are not shown here. They are described using coordinates in three-dimensional space and are best edited and displayed in the corresponding tool.

6.3 Dynamic Aspects

The main focus of this model is behavior, so besides the static view the description of the system's dynamic is of major concern. In the stream-based model the semantics of a component were defined by a stream processing function, which can be described using any mathematical theory and notation. AutoFOCUS uses automata as the primary description of a component's behavior. The appropriateness of automata can also be captured formally, as there is an isomorphism between strongly causal stream processing functions and Moore machines[4].

In our model the automaton will be the most important and flexible description technique as well, although some details are different from the automata used in AutoFOCUS. As experiments with an early version of the model indicated that certain problems are more complicates to express with an automaton than necessary, the meta-model is built to support alternate behavior specification techniques. This can be interpreted as the integration of small DSLs[5] for the solution of subproblems.

[4]The construction of a stream processing function from a state machine is described in [Bro97]. For the opposite direction the core idea is to relate the states to (sets of) finite prefixes of streams.

[5]Domain Specific Languages

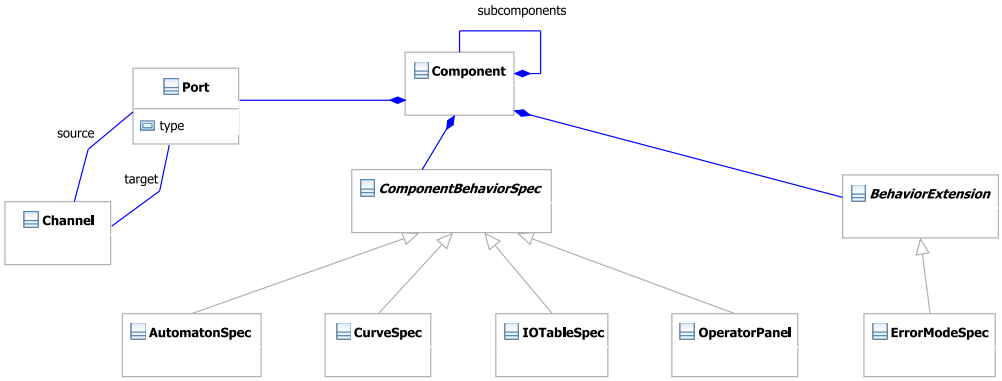


Figure 6.6: Meta-model for semantic component interface definition.

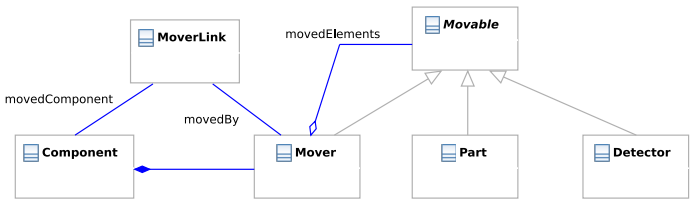


Figure 6.7: Meta-model for the application of motion/transformation.

The high-level view of the meta-model for behavior definition thus was chosen as shown in Figure 6.6. A component is defined by a *ComponentBehaviorSpec* with various implementations from which the automaton is only one. Additionally, the model supports the augmentation or modification of behavior by the so called *BehaviorExtension*. The idea is that a (non-composite) component is defined by exactly one *primary* behavior specification, but may have multiple orthogonal behavior extensions. This is roughly similar to the idea of *aspect-oriented programming* [KLM+97], where certain cross-cutting properties (called aspects) are specified separately. The *subcomponents* aggregation and the *Channel* class is explained in the context of composition in Section 6.4.

The following subsections explain the automata in more detail and briefly describe alternate specification techniques explored. Then, the behavior extension is explained using the example of orthogonal error specifications. Before describing the automata, the integration of motion into the model has to be explained.

6.3.1 Motion Application

As explained in Section 6.2.2 the operationalization enforces the shapes of parts and detectors to be static while their position may be changed over time based on the input observed, *i.e.*, the component's behavior. To describe this motion over time, movers are used. This is different from the modeling theory from Chapter 5, where movers are only used to affect other components (generated and non-generated). Our solution (*c.f.*, Figure 6.7) is to make a mover manage multiple *Movables*, which are parts, detectors, and other movers. To enforce encapsulation, a mover may only be connected to movables belonging to the same component. The reason for making the *Mover* itself a *Movable* is to allow the description of complex composite motion by chaining multiple movers. So, the position of a *Movable* is affected by its mover (if any) and recursively by the movers higher in the hierarchy. This way the movers form a directed graph. As cyclic dependencies are not sensible, they are forbidden and the mover graph thus is a forest.

The correspondence to the operation which connects a component to the mover of another component (Definition 5.13) in the meta-model is the *MoverLink*, which attaches a mover to another component. The details of this operation are explained in Section 6.4.

Now we can bridge the gap to the stream-based model, which requires at each time the volume taken by each part and detector and the transformation exercised by the movers. At each discrete time step, the component's behavior function assigns a scalar value from $\mathbb{R}$ to each mover. How this assignment takes place is explained below for the automaton-based specification. This value may also be affected by the environment model (*c.f.*, Section 6.6), *e.g.*, set to zero in response to collisions. The axis of a mover transforms this scalar to an actual transformation from the set $\mathcal{T}$. For a given mover and time $t \in \mathbb{N}$ we denote this (modified) scalar by m_t and the transformation associated to it by the mover's axis as $T_A(m_t)$. Then the transformation associated to this mover is given by $T_M.t$ and defined by $T_M.t = T_A(m_t) \circ T_m.(t-1)$ for $t > 0$ and $T_M.0$ is the identity transformation. Thus the current transformation is the result of multiple small transformation steps. In the case of a continuous model of time, the inductive definition has to be replaced by an integration over the transformation. The volume for a part or detector is then easily defined based on its initial volume V_I as $T_M.t(V_I)$, *i.e.*, the initial shape is just transformed. We emphasize again, that the calculation of a transformation from the scalar depends not only on the axis used but may also depend on the actual part or detector (especially its position) and thus has to be carried out for each moved element individually. However, this inductive definition simplifies the implementation of an evaluator for the model, as the positions at any time can be calculated from the positions from the previous time step.

6.3.2 Automaton-Based Specification

To specify the behavior of a component, we use a variant of automata. They consist of discrete control states, state variables, and transitions. Control states are connected by transitions, which model discrete events in the system. The interpretation is that time passes by while the component

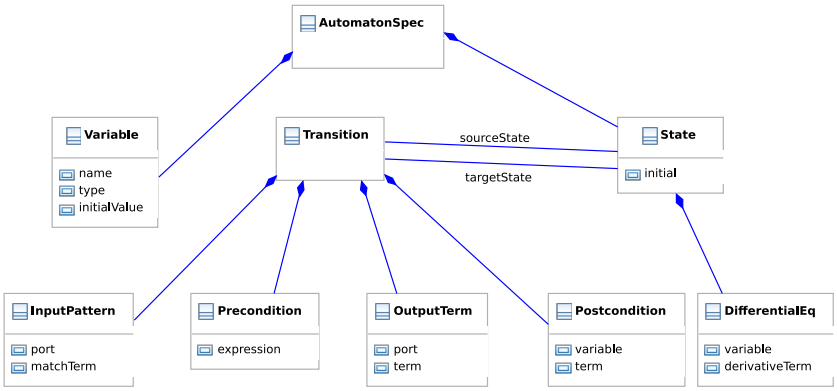


Figure 6.8: Meta-model for the automaton used.

is in one of the states, while transitions happen instantaneously. As we require strong causality for our components, the correct automaton model would be a Moore machine, where the output only depends on the current state and thus the new input can not affect the output. However, to simplify notation we annotate the transitions with outputs depending on the inputs, which corresponds to Mealy machine. Often this allows for a more compact automaton as Moore machines often require artificial states for buffering output. To ensure strong causality for the Mealy model, the automaton works on the input of the previous computation step, so the delay is realized by a one element buffer at the input ports[6]. This trick also simplifies the integration of weakly causal components as explained in Section 6.1.3, as for these we just have to disable this buffering, but can stick with the same automaton model.

Basic Automaton Model Exactly one control state has to be marked as initial. Analogously, the (typed) state variables require an initial value. During execution exactly one control state is *active*, at the beginning the active state is the initial state. The active state and the current valuation of the state variables together are referred to as the state of the automaton.

Transitions connect states and can contain multiple input patterns, preconditions, output terms, and postconditions. For the description of these elements (which are all optional) a term language based on the type system is used. Internally, all these elements are described with respect to a meta-model of terms, but due to the complexity of this part of the meta-model and the low relevance for the overall model, we do not provide more details on the internal representation, but only explain the textual syntax used. Examples are found towards the end of this section where the syntax is shown in an example, and in all example models discussed in this thesis.

[6] An equivalent solution is to perform buffering at the output ports or the connecting channels, as long as the delay from any output port to the input accumulates to exactly one discrete time tick.

Input patterns are bound to an input port of the component and can contain any match term (these are the same match terms used for function definition) which is valid for the type of the port. The match term may also contain variables, which are usually written using an underscore followed by an uppercase variable. The notation used is based on CSP [Hoa85] and uses a question mark to separate the port's name from the match term. The absence of a match term indicates the absence of a message (the ϵ-message) for this port. If the match term contains variables, their valuation used for matching is made available to the other elements of the transition.

The preconditions are boolean expressions that can depend on the state variables and the variables from the match terms of the input patterns. Output terms are used for sending messages and consist of the name of the component's output port to send the message at, followed by a (CSP style) exclamation mark and the term whose evaluation results in the value being sent. The postconditions provide for each variable the term which determines the new value of the variable. These are written as assignments using the equal sign. The terms in both output descriptions and postconditions may depend on state variables and variables from the input patterns, and must be of the correct type for the affected port or state variable.

Evaluation of the state machine is best explained step-wise (inductive). At each time tick, the transitions leaving the active state are evaluated based on the current inputs for the component and the values of the state variables. A transition is called *ready*, if for each input port referenced from an input pattern the corresponding match term matches the current input and all preconditions evaluate to *true*. From the set of ready transitions one transition is chosen nondeterministically. For this transition the the output terms are evaluated and assigned to the corresponding output ports and the terms of the postconditions determine the new values of the state variables. Finally, the state at the target end of the transition becomes the new active state. If for an output port no output term is provided by the transition, either the empty message ϵ is sent (signal port) or the value from the previous step is sent again (state port). Similarly, state variables without corresponding postconditions preserve their value. If at a time step no transition is ready, the active state does not change and all output ports and state variables are treated as if an empty transition was executed. At each time step only one transition is taken.

Pseudo-Hybrid Extension In pure software systems often only relative time is important, *i.e.*, the ordering and dependence of events to each other. Models of mechatronic systems, in contrast, often require the actual real time passed by as the progress of physical processes and movements depend on this. To capture such timing properties and the flow of time, there are special automata. *Timed automata* [AD94] allow the definition of clocks, which are real valued variables which are increased continuously with the progress of time. These clocks can be included in the guards (preconditions) of transitions and reset to a value of zero. A more general model are *hybrid automata* [Hen00], which allow progress of each variable to be defined using a differential equation in each state. Timed automata can be embedded in the theory of hybrid automata by setting each variable

x corresponding to a clock to constant progress, *i.e.*, using $x' = 1$ as differential equation[7]. While many properties of timed automata, such as reachability, are decidable, for hybrid automata they are only decidable for specific subclasses with severe limitations regarding the differential equations used. Even worse, many hybrid automata can not even be simulated without errors [Mos99].

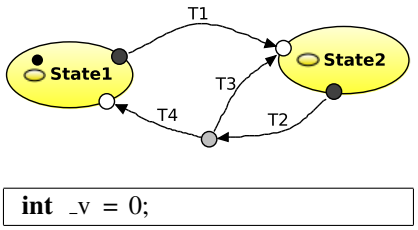
Both timed and hybrid automata use a continuous model of time, while our time model is discrete. However, the need to increase the value of a variable based on time is common and cumbersome to write down as the increase would have to be added as a postcondition to every transition. So we incorporate the idea of hybrid automata in our automaton model and allow the augmentation of states by differential equations. These equations may only be defined for variables of type *double* and the variables associated with spatial motion as explained in the next section. The term describing the derivative value may depend on all state variables and the notation is based on the mathematical one, *i.e.*, _x' = 1 captures the notion of a clock which increases synchronous to real time.

It is important to remember that our model has no continuous time and thus these differential equations can not be evaluated faithfully, especially the transition conditions are evaluated only at discrete time steps, so the time frame where a certain transition would be ready might be missed. Instead the equations can be seen as a kind of *syntactic sugar* but also help in obtaining a clear separation between discrete and continuous change, thus we refer to our automaton as *pseudo-hybrid*. The interpretation and evaluation of the differential equations is performed in discrete time. For this, at every time step prior to the selection and execution of a ready transition the state variables are updated based on the equations belonging to the active state. A useful side-effect is that the model can be formulated independently from the real time corresponding to a discrete time step, as the time only appears implicitly in the differential equation. Other formulations usually require some variable δ which contains the amount of time passed since the last time tick and can be used as a factor. Thus, the actual granularity of the simulation time (the duration between two consecutive ticks) can be chosen independent from the model and even be changed during evaluation of the model to sample certain sections of time with higher precision (lower duration between ticks).

When modeling, it must be kept in mind that this simple discrete evaluation of differential equations can lead to approximation errors and numerical instability. However, often this can be neglected at our level of abstraction and choosing the time granularity fine enough can compensate for most of the errors[8]. Thus, the differential equations behave just as in hybrid automata for most of the common cases, without causing the problems common there. Especially simulatability and decidability of properties such as reachability are not affected as we still deal with a discrete system.

[7] We use x' for the derivative with respect to time instead of $\dot{x}$, which is commonly seen in physics and engineering. The reason is that x' is easier for a user to write in a tool implementation and there is also no danger of confusion as all derivatives in this thesis are with respect to time.

[8] Another means to compensate for some of the errors would be the application of a better numerical integration method. The method chosen here corresponds to the so called *Euler method*, while more advanced methods, such as the *Runge-Kutta* or *Bulirsch-Stoer* methods could resolve some of the errors.



int _v = 0;

(a) State graph

T1	**in:**	run?**true**; ready?_X
	pre:	_X > 5 && _d
	out:	state !**true**; vel !_v+1
	post:	_v = _v+1
T2	**in:**	run?**false**
T3	**pre:**	_v > 7
	post:	_v = _v+1
T4	**pre:**	_v > 12

(b) Transition specification

Figure 6.9: Graphical syntax used for automaton description.

Integration with the Spatial Model The description of the automaton so far did not contain any link to the geometric parts of the model, specifically the detectors, parts, and movers. The first link is the reaction to activations of detectors of the component, which indicate the collision of some part with the volume of the detector. This corresponds to the $\mathcal{A}(D)$ part on the input side of the semantic function for a spatio-temporal component (*c.f.*, Section 5.2). In the automaton this is realized by mapping each detector to a read-only state variable of type boolean which then can be used in the preconditions, output terms, and postconditions of the automaton's transitions. This variable is *true* in case of detected parts (collisions) and *false* otherwise.

The other link are the movers, which affect the positions of parts, detectors, and components. As explained in Section 6.3.1, during each tick a scalar value has to be assigned to each mover of a component. The actual transformation is then calculated based on this value and the axis used. This scalar value, which corresponds to the amount of motion, has to be set by the automaton. For this, each mover is assigned a variable which can be used as the target (left hand side) in a state's differential equations. The reason to allow the assignment only in differential equations is the intuition that motion is applied as long as the component is in a certain state. Furthermore, motion automatically respects the amount of time passed by, *i.e.*, the component may not set a position to an absolute value and thus violate spatial continuity. As the environment (*c.f.*, Section 6.6) may change the scalar assigned to the mover, this value is also made available as a read-only double variable for the transitions. This way the automaton can decide in a state how fast the mover should be operated and then check in the transition (at discrete time ticks) how much motion actually was applied. Using this mechanism, a controller may detect blocking of a moved part. In real systems this is usually realized by separate sensors or by observing the power consumption of the electric drive. An example of the later case is the squeeze protection for a car's power window.

Textual and Graphical Syntax The graphical syntax used for the automata is based on the state graphs commonly found. Figure 6.9 gives an artificial example including both the state graph with the declared state variables as well as the transition conditions. The control states are rep-

resented by the ellipses, the initial state is marked by a black dot within the ellipse (in this case called *State1*). The only state variable (called _v) is declared using the type, its name, and the initial value.

The transitions are depicted by curved arrows connecting the states with the transition names written beside the lines. These names are referenced from a separate table providing the details of the transitions. For each transition the input patterns (in), preconditions (pre), output terms (out), and postconditions (post) are listed. If any of these elements is empty, it is omitted from the table. In the example, transition T1 is ready if the component receives the value *true* on its *run* port and any value larger than 5 on its *ready* port. For the later condition a local variable _X from the match term is used in the precondition. Additionally, the value of _d has to be *true*. As it is not a local variable we can assume that _d is associated with a detector. If the transition fires, the value *true* is sent on the port *state* and _v+1 on the *vel* port. Additionally, the value of _v is increased by 1. It should be noted that the _v at the left hand side refers to the new value of _v, while the right hand side references its old value[9].

The transitions do not directly connect states but *interface points* belonging to the states. The black ones indicate outgoing points, the white ones are transition targets, and the grey ones do not belong to any state and are called *local interface points*. We will only explain the local interface points here, as the white and black ones are only relevant for hierarchical automata, which are ignored in this thesis to keep the presentation free from too much detail. Still, the tool prototype (*c.f.*, Chapter 7) supports hierarchy and thus all diagrams shown will include these interface points. Local interface points are just syntactic sugar to avoid redundant specification of transition conditions. The interpretation is that any possible (loop-free) sequence of transitions connected by local interface points will be considered during the search for ready transitions. Such a transition sequence will be treated as a single sequence by concatenating all its conditions. So instead of the transitions *T2*, *T3*, and *T4* we could have used two transitions *T2T3* leading from *State2* to itself and *T2T4* leading from *State2* to *State1* and having the following conditions:

T2T3	**in:**	run? **false**
	pre:	_v $>$ 7
	post:	_v = _v+1

T2T4	**in:**	run? **false**
	pre:	_v $>$ 12

This is also an example for a nondeterministic automaton as both transitions are ready if _v $>$ 12 (and run? **false**). The use of local interface points is especially useful when encoding error detection and handling. For this case often every state contains transitions with complex conditions leading to a state that initiates the error handling. Usually, the condition is the same for each of these transitions and can be *factored out* using a local interface point.

[9] From a mathematical perspective this looks like the unsolvable equation $v = v + 1$ and for clarification two different names should be used, *e.g.*, v_{new} and v_{old}. However, the simplification of using the same name for the variable is common in most programming languages and thus adopted here.

6.3.3 Alternate Specification Techniques

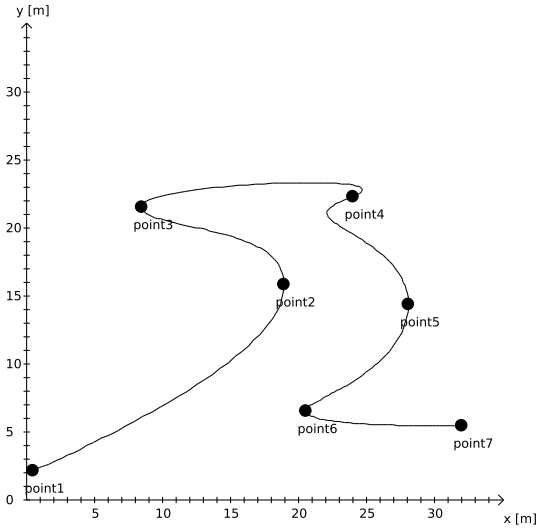
Besides the automaton based specification, our framework supports the usage of alternative specification techniques. While state machines are the most generally applicable, other techniques can be more efficient for certain behaviors. Following, three specification techniques are described briefly, which were subject of research and are also implemented in the tool prototype (*c.f.*, Chapter 7). However, only the operator panel specification has been used in the case studies described in Chapter 8, the other two have only been used in small example models so far.

Graphical Function Specification There are some functional dependencies, which are hard to formulate as a mathematical equation, but easy to sketch on paper in terms of a function graph. Prominent examples are the characteristic maps used in car engine controllers. In factory automation a common problem is to follow a given path with a robotic gripper or a tool. A common configuration for two-dimensional paths consists of two separate axes in x and y direction which are powered by independent drives. An example of a path is shown in Figure 6.10(a). Such a path is realized as a function mapping from one input value (usually interpreted as relative time) to the x and y coordinates.

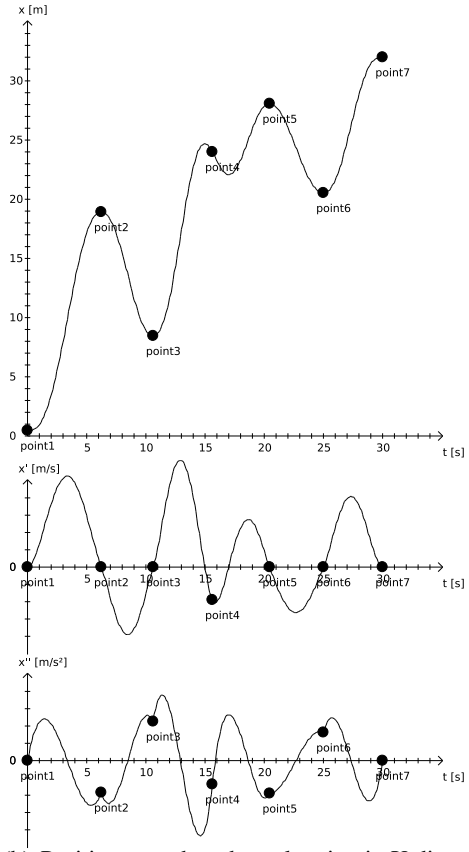
The challenge in designing the path is that the velocity and especially the acceleration for both axes should be kept within certain bounds to reduce the wear of the drives and keep the precision as high as possible. In Figures 6.10(b) and 6.10(c) the position, velocity (first derivative), and acceleration (second derivative) curves for the x and y axis are shown. While for the y axis the acceleration has been reduced by careful manipulation of the curve parameters, the x axis still has peaks of high acceleration.

To support the definition of such motion paths, a curve-based specification is provided. A curve is defined by interpolation points for which the x and y coordinates, the time value, and the first and second derivatives of x and y with respect to time can be defined. The missing values are calculated by piecewise interpolation with 5th degree polynomials. By providing the curves based on these interpolation points, the effects of parameter changes, especially of the derivatives, can be explored. The curves from Figure 6.10 are taken from the actual tool implementation (*c.f.*, Chapter 7) where these diagrams are embedded in an editor.

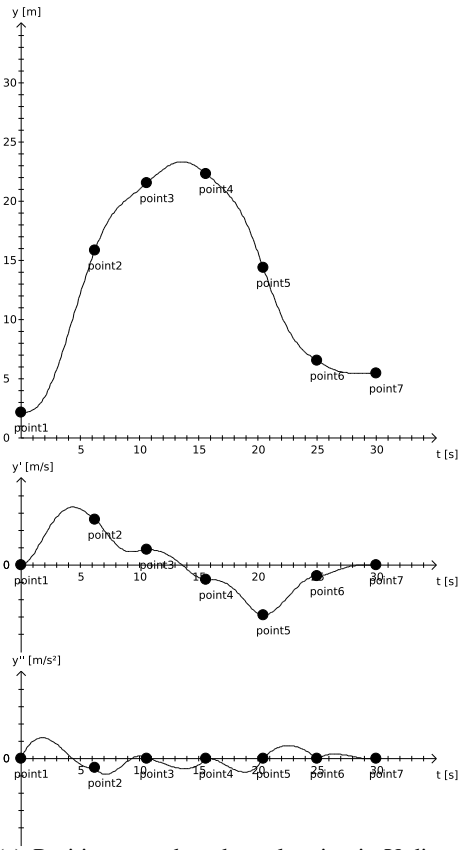
The semantic embedding into the theory is fairly simple. The specification can only be used for components with a single (double valued) input and outputs for any of the position, velocity, or acceleration values. At any (discrete) time the value at the input port is read and interpreted as the time value. The values at the output ports are then set according to the function evaluation regarding the time value. Technically, this is performed by translating the curve into an automaton with one transition for each spline segment. The precondition limits the transition to inputs within the range of the segment, while the output is calculated based on the segment's defining polynomial. The automaton translation has the benefit that all operations available for automata are applicable to these curves as well.



(a) The X-Y position curve



(b) Position, speed, and acceleration in X direction



(c) Position, speed, and acceleration in Y direction

Figure 6.10: Example for graphical function specification.

User	Status	Command	Annotation
on	off ϵ^*	on	When the user input on is received and the last status signal has been off, the monitor sends the command on to the station.
off	on ϵ^*	off	When the user input off is received and the last status signal has been on, the monitor sends the command off to the station.
.	$\epsilon\{5,5\}$	req	When the monitor receives 5 time intervals no status signal (ϵ), it sends a request (req) to the station.
.	on|off	.$\{0,2\}$ ack	When the station signals a status (on or off), the monitor has to confirm this (ack) within the next three time intervals.

Figure 6.11: Example for a stream-based I/O table.

Stream-Based I/O-Tables Contrary to state-based formalisms, such as the automaton model described earlier, requirements are often formulated in a more sequence oriented fashion (*e.g.*, "after message A has been received four times, message B should be sent at least twice"). An approach to directly capture such sequence-based requirements are *stream-based I/O tables*, which were introduced in [TH09]. As these tables describe a system by a stream-processing function, they are a perfect match for our model.

For a given component, its input and output ports define the columns of the table, while the different rows of the table specify the different requirements on the I/O behavior of the component. The expressions in the input cells define input patterns by describing the last messages that have been received on the input ports until a time t. The expressions in the output cells define corresponding output patterns, *i.e.*, messages that must be sent in the following time intervals whenever the input patterns are fulfilled. To preserve the mapping between the informal requirements and the formal specification, the table comprises an additional column for annotations. The requirements are grouped into different segments separated by double horizontal lines. Different segments must always be fulfilled simultaneously.

The usage of these tables is best illustrated by an example, which we recapitulate from [TH09]. We specify a monitoring component (just called *monitor* in the remainder) via which a single station of an automation system can be logged in and out. The monitor is used to decouple the user interface from the external station. The monitor has two input ports, *User* to receive the user input and *Status* to receive the status of the station, and one output port *Command* via which the monitor sends commands to the station. The ports *User* and *Status* are of type *OnOff*, and *Command* is of type *OnOffAck*, where the types are defined as follows:

```
OnOff    = on | off ;
OnOffAck = on | off | ack | req ;
```

The stream-based I/O table specifying the component is given in Figure 6.11. The first segment formalizes the logging in/out of the station, while the second segment formalizes the status control. To clarify the presented table-based specification, we explain some of the expressions used in the

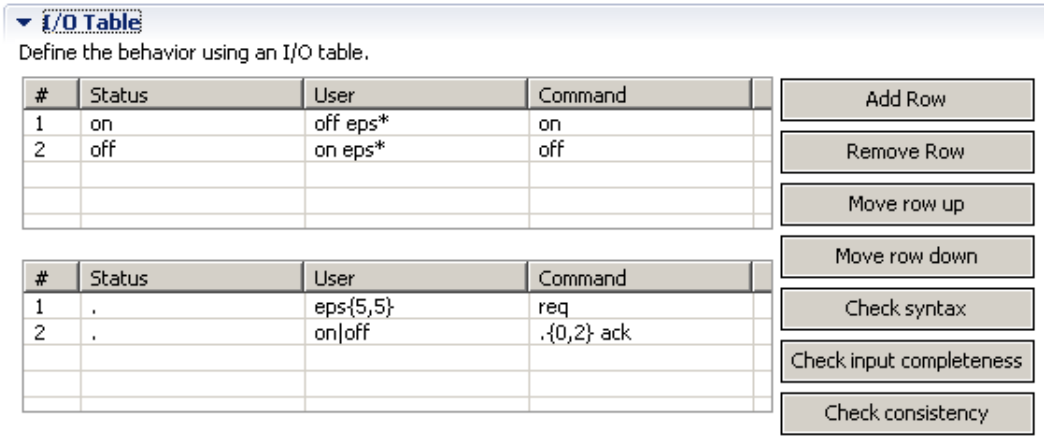
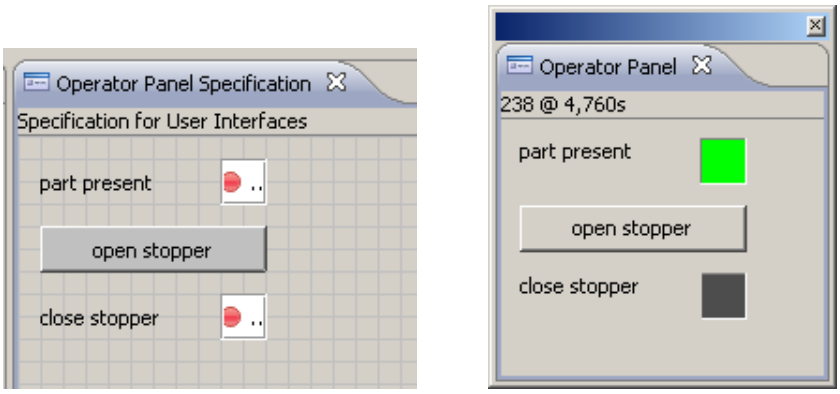


Figure 6.12: The example from Figure 6.11 in the table editor.

table. The regular expression "on ϵ^*" models that the last signal received on port *Status* until time t has been on. The time intervals after that, only ϵ – meaning no signal – has been received. The notation $x\{min, max\}$ is used to describe that a certain signal or regular expression x occurs min to max times in a row. Thus, the expression "$\epsilon\{5, 5\}$" formalizes that in the last five time intervals no signal is received on the respective port. The character . declares that any not further specified signal of the corresponding type (including ϵ) is received/sent in the respective time interval. The expression "$.\{0, 2\}$ ack" describes that – starting from time interval $t + 1$ – first for 0 to 2 time intervals any signal can be sent, but at latest in the third time interval, ack has to be sent.

The precise semantics of these tables as well as algorithms for consistency checking are described in [TH09]. Support for table editing as well as the checking algorithms is integrated into the tool (*c.f.*, Chapter 7) as implied by Figure 6.12. However, the I/O tables currently lack support for local variables and description of (pseudo) continuous behavior similar to the differential equations used in the automaton based specification. If these problems are solved, they also can be used for the integration with the spatial parts of the model by mapping detectors and movers to ports or local variables as with the automata. The full integration of these techniques, however, is the subject of future work.

Operator Panels A different view on components is the starting point for operator panel specifications. These are used to define the input options provided to a machine operator and are described by common user interface elements. The specification maps these user elements to values for the component's ports, thus the user interface and especially the user's actions can be interpreted as just another component. As all possible user actions are considered, the component is highly nondeterministic.



(a) Specification view (b) Simulation view

Figure 6.13: Operator panel based specification.

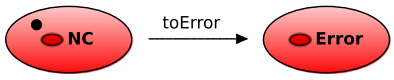
Figure 6.13(a) gives an example of a simple operator panel specification. The button labelled by "open stopper" is described by additional terms which associate it with the *open* port of the specified component:

pressed:	open!**true**
released:	open!**false**

Here the same syntax as for the automaton is used, so the component will send a *true* signal on the *open* port while the button is pressed, and *false* if it is in the released state. The small squares are indicator lights, which visualize the current value present on the input ports of the component. For example the upper one is specified by the following terms.

close_stopper ?**true**	→	green
close_stopper ?**false**	→	grey

So, the lamp will be green if the port *close_stopper* receives a *true* signal and grey otherwise. Obviously, display elements, such as these lamps, do not contribute to the component's behavior. The reason to include them is the second application for which these specifications are used. When simulating the model, the operator panel can be displayed as a *normal* user interface which can be used to interact with the simulated machine (*c.f.*, Figure 6.13(b)). In this case the behavior of the component is no longer nondeterministic, but defined by the user's actual interaction. Besides buttons and lamps, several other user interface elements are supported. These elements as well as all options for mapping these element's states to the component's ports is described in [Döb09], in which context also most of the implementation was performed.



NC		
Error	**input filters:**	goal = goal + 1
	output filters:	speed=0; running=!running

Figure 6.14: Example of error mode specification

It is important to differentiate these operator panels from user/usage models. Our specification is focussed on the description of the user interface itself to define all possible user inputs. It can also be exploited for further activities, such as simulation and automatic generation of the user interface of the real system during its development. User modeling, in contrast, describes possible user inputs as well, but also includes a model of common input sequences often augmented by probabilities and possibly depending on the system's feedback. Thus, while our model can only select user inputs nondeterministically, these models can generate user input which matches the expected probability distribution of real user inputs. A possible direction for future work would be the extension of the operator panel specification to also include user/usage modeling.

6.3.4 Behavior Extension and Error Modeling

As explained at the beginning of Section 6.3, a component's behavior can be complemented and modified by additional specifications, which we call behavior extensions. This allows to use different description techniques for orthogonal aspects of the behavior. One example for an extension is the specification of the material flow interface, which is used to describe the entries and exits of a component as described in Section 5.3.1. As these are usually only required for few components, these elements were moved to a separate specification to simplify the component interface slightly. The material flow interfaces are described in more detail in Section 6.5.

The second application of behavior extension is in the area of model-based fault injection. A significant part of our models is used to describe actors and sensors which are potentially error-prone. A sensor could send wrong values because of an electronic defect or dirt and dust on its detection area. A motor might stop due to a broken wire in its power supply or turn slower because of wear. It can be useful to describe possible errors for components representing hardware, as these components can be used to test whether the controller is capable of ensuring safety even in the presence of hardware defects.

During development it is often desirable to switch between a version without hardware errors to analyze the normal behavior, and a version including the errors to inspect reaction to and handling of hardware problems. For this reason an orthogonal description technique as described in [BH09] has been chosen and implemented as a behavior extension. In this formalism, different error modes can be differentiated and their dependency described as a state machine. An example is given in Figure 6.14, which consists of the two states *NC*, being the initial normal condition, and the *Error* state. The single transition indicates that the component can go from normal mode to the error mode,

but not back, so repairing is not modeled. The transition *toError* can be guarded by expressions similar to the automaton specification, but in this case it is left unguarded which indicates that the component may switch to the *Error* state nondeterministically at any time.

Each error mode is described by sets of input and output filters which are used to modify the input and output streams of the component. The behavior of a component is then defined at a given time by the following steps. First the state machine is executed to potentially switch to another state. Then the component's inputs are read and modified by the input filters. These modified inputs are then handed to the actual specification of the component (which could be an automaton specification or any other of the specification techniques described before) which in turn may update its state and returns new outputs for the component's ports. Before writing these values to the output ports, they are again modified by the output filters. In the example of Figure 6.14 the mode *NC* has no filters and thus does not modify the behavior. The state *Error* has an input filter which increases the value read on port *goal* by one and thus simulates a calibration error. The output filters set the value of the *speed* port to 0 and negate the value of the *running* port, which might correspond to a defect of the internal hardware controller.

The main advantage of this filtering approach is its compatibility with the stream-based component model, which allows the orthogonal application and thus the realization as a behavior extension independent of the specification type used for the primary behavior description. Of course the limitation to filters makes certain behavior changes impossible or at least very complex to specify, however, as explained in [BH09], all effects commonly encountered in real systems can be expressed by rather simple filter expressions.

6.4 Composition

Composition is the combination of multiple components into a single component. This allows to handle complex problems by dividing them into manageable parts and assembling the solution from them. Furthermore, existing building blocks can be easily integrated into new designs by composition. As the semantics of component composition has been discussed in depth in Section 5.2, we focus on its representation in the meta-model and the graphical syntax here.

6.4.1 Data-Flow Composition

To express a data-flow connection between components, their input and output ports can be connected by a *channel* (*c.f.*, Figure 6.6). Ports may only be connected by a channel if they are of the same type[10] and each input port may only be connected to at most one incoming channel. Contrary,

[10] Actually it would be sufficient to require compatible types, *i.e.*, the carrier set of the type of the sending port must be a subset of the type of the receiving port. As our type system is rather simple and does not currently support this kind of sub-type relation, we stick with the more strict requirement of same types here.

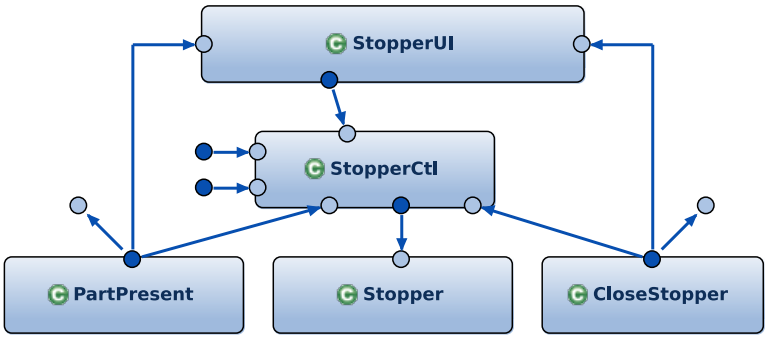


Figure 6.15: Graphical syntax used for data-flow composition

output ports may lead to multiple outgoing channels, which can be interpreted as data duplication.

The meta-model shown in Figure 6.6 has an aggregate association called *subcomponents*, which allows a component to be described by a set of other components. Although not visible from the figure, a component may either be composed from other components *or* described by a behavior specification, which is enforced by corresponding constraints. Behavior extension may be used parallel to composition. The subcomponents of a composed component are *wired up* by channels as described before. Additionally, ports of the subcomponents can be associated with the ports of the surrounding component. By this the syntactic interface of the composed component is formed by a subset of the ports of all its sub-components. The parts, detectors, and movers of the composed component are just the union of the respective elements of the subcomponents. Using composition as described here can express both the parallel composition (Section 5.2.2) and data feedback (Section 5.2.3).

The graphical syntax is based on the rectangle representation already known from Figure 6.5. As shown in Figure 6.15, channels are depicted as lines with arrow heads, connecting two ports. The four ports drawn on the background (not on the border of any component) are the ports of the composed component and the channels describe the association of these *outer* ports with ports of the subcomponents. The five components shown here form the single component from Figure 6.5. Thus the four *free* ports correspond to the four ports of the *StopperUnit*. As it is not relevant here, we will not go into the details of which ports are connected. Where necessary, the images are augmented by labels to indicate the names of ports.

6.4.2 Spatial Composition

A composed component may introduce additional (static) parts, and may apply a fixed transformation to its subcomponents (corresponding to the positioning operation from Section 5.2.4). As this is performed by definition of coordinates and transformation in three dimensions, it is best applied in a tool.

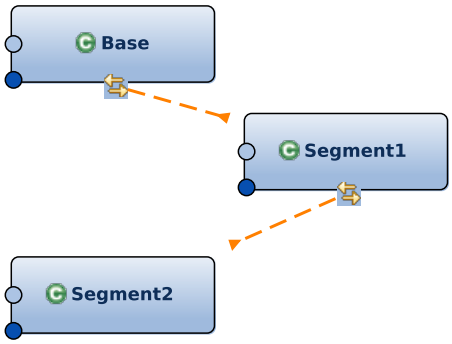


Figure 6.16: Graphical syntax used for spatial composition

Furthermore, a component can be connected to the mover of a sibling component (*c.f.*, Section 5.2.4) during composition, which makes the component follow every transformation of the respective mover. Graphically, this is represented by dashed lines as shown in Figure 6.16. The line connects a component with a mover (symbolized by the small double arrow icon) of a sibling component. In the meta-model this is realized by the *MoverLink* model class shown in Figure 6.7. As kinematic loops are not supported, no directed cycles may be created by mover links. Additionally, only components *on the same level*, *i.e.*, having the same super component, may be connected by mover links. Of course, spatial composition may be combined with data-flow composition, as shown in examples from the case studies (Chapter 8).

6.5 Material Flow and Generated Components

An essential aspect of our model is the treatment of material and its interaction with the system's components. As described in Section 5.3 we use the same component-based description technique as for the system itself when modeling material. This is done to not increase the number of model elements further and also because the material can be as complex as the system itself. The main difference is that for a material component there can be multiple instances in the simulation, and instances have to be created and destroyed dynamically. To control the insertion and removal of material over the life-time of the system so called *entries* and *exits* are used. In the context of the system they describe, material is relevant for the system after passing through an entry and before leaving through an exit. In the context of a simulation, the entries can be interpreted to *generate* material, while the exits *destroy* it. The second ingredient to allow material simulation is the definition of interaction between material and the components comprising the system. The modeling element used to express this is the *binding*.

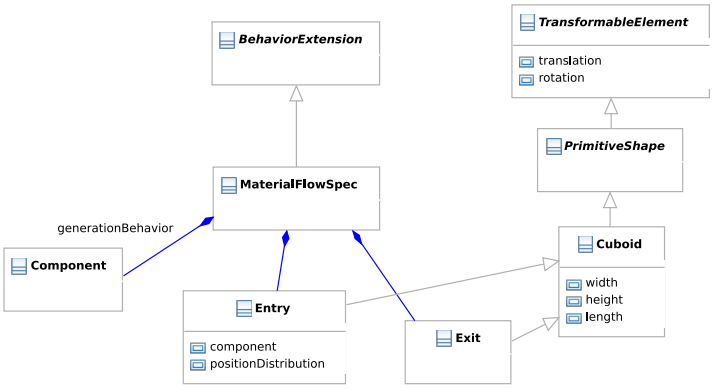


Figure 6.17: Meta-model for definition of material flow

6.5.1 Entries and Exits

The material flow for a component is defined by a behavior extension called *MaterialFlowSpec* shown in Figure 6.17. The interface of the material flow is defined by *entries*, which are used to introduce new (material) components to the system, and *exits*, which are used to remove them. Both are associated with a spatial volume to describe where components are inserted or discarded, as indicated at the right side of the diagram. Each entry contains a component name, which describes the type of material (component) being generated by it, and a distribution which is used to slightly perturb the initial location of the generated material relative to the entry's position. The exit affects material of any type when it enters its volume. The reason to extract material flow definition to a behavior extension is that usually only few components directly define their material flow. Especially primitive components, such as drives, stoppers, or sensors, do not have a sensible notion of material flow, but rather more complex composed components, such as transportation systems or entire automation systems may benefit from the definition of the material flow.

While the entries and exits define something similar to a syntactical interface, the semantics, *i.e.*, the exact times at which material is inserted or extracted, is defined by a component (association *generationBehavior*). This component has to provide output ports of *boolean* type corresponding to each of the entries and exits (which is checked by constraints not shown in the meta-model diagram). The interpretation of the output streams of this component is that an entry inserts a new component corresponding to its associated type, whenever a *true* message is emitted on its corresponding port. The position of the new component is the same as the position of the entry, but may be modified by a (randomly distributed) perturbation. Similarly, an exit removes all (material) components from the system whose centers are inside of its spatial volume, if and only if *true* is sent on the corresponding port. The benefit of reusing components to define the semantics of the material flow is that no new modeling elements are required and that the full machinery introduced for components is available. This includes the possibility to construct the component from multiple subcomponents, define the components by one of the different behavior specification styles, or allow these components to have

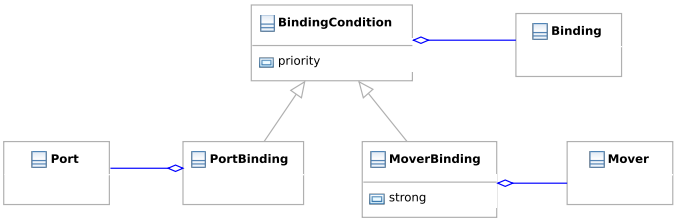


Figure 6.18: Meta-model for definition of material bindings

spatial elements. From the spatial elements mostly the detectors are used, which allow the material flow to depend on spatio-temporal properties. A common example is a restriction to generate a new material element only if the space is not occupied by another (previously generated) one. This can conveniently be checked by a detector.

6.5.2 Binding Conditions

The material flow specification defines where material enters and leaves the scope of the system. At least equally important is the interaction between components of the system and the material. The detection of material is performed by detectors and has been described before. When affecting the material, however, two questions arise. The first is *when* interaction does occur, the second is *what kind* of interaction it will be. To determine, when material is in *interaction range*, we use bindings, which are similar to detectors in that they have a volume (modeled by initial shape and current position) and are checked for collisions with parts. The difference is that bindings can be used in selection predicates to select material that is (temporarily) connected to a mover or a part.

The realization in the meta-model is shown in Figure 6.18. The bindings are used in binding conditions which exist in two *flavors*. A *PortBinding* associates bound material with an input or output port, while a *MoverBinding* creates a temporary connection to the given mover. A component is said to be *bound by* a binding condition, if its parts collide with all of the bindings associated with the condition[11] (this corresponds to the *collision selection predicate* introduced on Section 4.2.2). Components bound by a mover binding are affected by this mover while being bound. The exact motion applied also depends on possible collisions in the environment and is discussed in Section 6.6. For mover bindings the volumes covered by the bindings usually correspond to the surface of some friction-based transportation belt or the two jaws of a gripper. The later case is an example where only material colliding with both bindings should be affected.

[11] The actual implementation supports two sets of bindings for a binding condition. A component is then bound if it collides with all bindings from the first set (inclusion) and none of the second set (exclusion). Even more complex model supporting arbitrary predicates over the bindings can be imagined. However, none of the systems encountered and modeled so far required more than simple inclusion, so we stick with the simplest model here.

While a component is bound by a port binding, a temporary connection between the port of the system's component and the port of the same name (if any) in the component is created. A binding could for example model the sensor range of an RFID scanner. Components then should have a port named *rfid* on which the numeric value of its tag is sent. When material is in the scanner's range, the scanner component connects to the *rfid* port and can read the tag value.

For both kinds of binding conditions the case of multiple bindings has to be resolved. How this is done, is explained in the next section. In the textual syntax we annotate ports and movers with binding conditions by appending the **bound by** keyword to the port's or mover's declaration followed by the names of the binders belonging to the condition. In the case of movers we also use **weakly bound by** and **strongly bound by** to indicate the value of the *strong* attribute (details discussed in Section 6.6).

6.6 Environment Semantics

The extended spatio-temporal components introduced in Section 5.3.1 allow for some freedom with respect to the exact semantics of the interaction between generated components. One question is, which (generated) components are moved and how, as Invariant 5.15 only requires motion to be applied in a way that no collisions occur. Similarly, Invariants 5.16 and 5.17 require that communication on external ports may only occur while they are bound, but do not explain what happens in the case of multiple components being bound to the same port. As we need these details during simulation and interpretation of the model, we will deal with them in this section.

6.6.1 Bindings with Movers

The invariants from Section 5.3.1 require components (more precisely their parts) to never overlap and motion of generated components be based on a mover it is bound to. To avoid collisions, a component may also not move during a time step even if it is currently bound to a mover. Two questions are left unanswered there:

1. What happens if a generated component is bound by multiple movers?
2. How is decided whether a generated component is moved or rather stopped to avoid collisions?

Our operationalization answers both of these questions. For this we employ a two-phase process in each simulation step. First, for each (generated) component the binding to be used (and thus the motion) is determined. In a second step we determine which of the components is moved and which not.

To determine the binding used in the first phase there are two precedence rules: a strong binding is always preferred over a weak binding and for equal *strength*, the binding with higher priority

is used. Strong bindings model fixed connections to the material (*e.g.*, by a gripper), while weak bindings indicate the possibility to slide, as for material on top of a belt conveyor, thus strong bindings will *win* in this case. The priorities can be used, for example, to express a higher friction coefficient compared to another binding. If there are multiple bindings with the highest strength and priority, the binding used is chosen nondeterministically. Such a situation can occur for instance on the hand-over point between two belt conveyors, where material is located on both belts at the same time. The exact motion applied to the material is very hard to determine, even for more detailed simulation models, but the nondeterministic resolution models the real system well enough in most cases. For generated components we also support the automatic binding to a *gravity mover* in the case that it is bound to no other movers. This mover advances the component downward, thus modeling gravity.

After the first step, the binding and thus the applied motion is known for all components. The goal now is to move as many of these components as possible without violating the collision invariant. For this, collisions are calculated between all components (or their parts) for both the current and the new (intended) position. As the collision invariant holds, no two components may collide at their old position. If a component does not collide with any other component for both the old and new position, it may clearly be moved. Contrary, if a component collides with a component at both the old *and* new position, it may clearly not be moved. For the remaining cases, it depends on the motion of the other component, *i.e.*, if it only collides for the new position of the component, it may be moved only if the other component is *not* moved, and vice versa. This leads to dependencies between components, where certain components may be (not) moved only if other components are (not) moved. In our operationalization the goal is to move as many components as possible. However, the problem can be reduced to the MAX-2-SAT[12] problem, which unfortunately is NP-complete and thus would make real-time simulation nearly impossible if this has to be solved at each simulation step. Thus, we only use a greedy solution of the above problem. As configurations of components leading to complex instances of the problem are rare, the greedy solution is usually sufficient.

Once we have determined which components may be moved and which not, the movement is propagated. This is where the weak and strong movers are needed. For a strong mover, the entire mover is stopped if a single component bound to it is stopped. So all other components bound to the same mover will not move in this simulation step as well. This is used to model, for example, a robot with a gripper which holds multiple material pieces. If only one of the pieces stops due to a collision, the entire robot arm is stopped, and hence all other material pieces as well. In contrast, the weak binding only stops the affected generated component. For example other material on a conveyor belt is still transported even if a single component bound to it is stopped, as the belt slides below the component.

[12] This problem is also known as maximum-2-satisfiability and consists of a 2-SAT formula (*i.e.*, a Boolean formula in conjunctive normal form with two literals per clause), for which we search for the variable assignment which violates the least number of clauses (or dually fulfills the maximal number of clauses).

6.6.2 Bindings with Ports

When using bindings with ports, besides the normal one-to-one connection case there are four relevant scenarios:

1. A component's output port is bound to the input ports of multiple generated components.
2. A component's input port is bound to the output ports of multiple generated components.
3. Multiple components' output ports are bound to the same input port of a generated component.
4. Multiple components' input ports are bound to the same output port of a generated component.

The n-to-n case is omitted as it can be easily derived from these four cases. Cases 1 and 4 are already answered by Invariants 5.16 and 5.17, which require an input value to be present if one is provided via a bound port. This is the same situation as when we connect multiple input ports with one output port: the transmitted message is just duplicated. More interesting are the cases 2 and 3 which are *multiple write* scenarios, where an input port receives messages from more than one output port. During composition we disallow this situation by removing an input port from the component's interface after connecting it. During simulation, this situation can not always be avoided. There are several possibilities for resolution, outlined next.

Invalid: The input port receives an invalid message represented by a specific symbol (such as $\perp$). This models the situation where multiple messages on the same bus or transmission frequency produce garbage, which can be recognized by the receiver (*e.g.*, based on check sums). The drawback here is the introduction of a new symbol which has to be included by types and respected by the components involved.

Chaos: The input port receives an arbitrary message (chosen nondeterministically) from the carrier set of its type. This models the situation where multiple messages on the same bus or transmission frequency interleave and produce a new valid message (*e.g.*, because all messages are valid).

Merging: The value on the input port is calculated by merging the messages of the output ports. A typical merging scheme is to establish an order on the messages and use the largest message with respect to this order. This scheme is, *e.g.*, used in message arbitration in the CAN bus [ISO03a]. The drawback of this solution is the need to define a merging scheme for all types used.

Nondeterminism: The message received is chosen nondeterministically from all messages offered. This resembles the situation where one of the output ports is best to receive (*e.g.*, based on position or signal strength), but our model is not detailed enough to decide which one. This is expressed by nondeterminism.

Priorities: We could also use priorities on the ports or bindings to model different signal strengths. Then the message from the port/binding with the highest priority would be used as input. However,

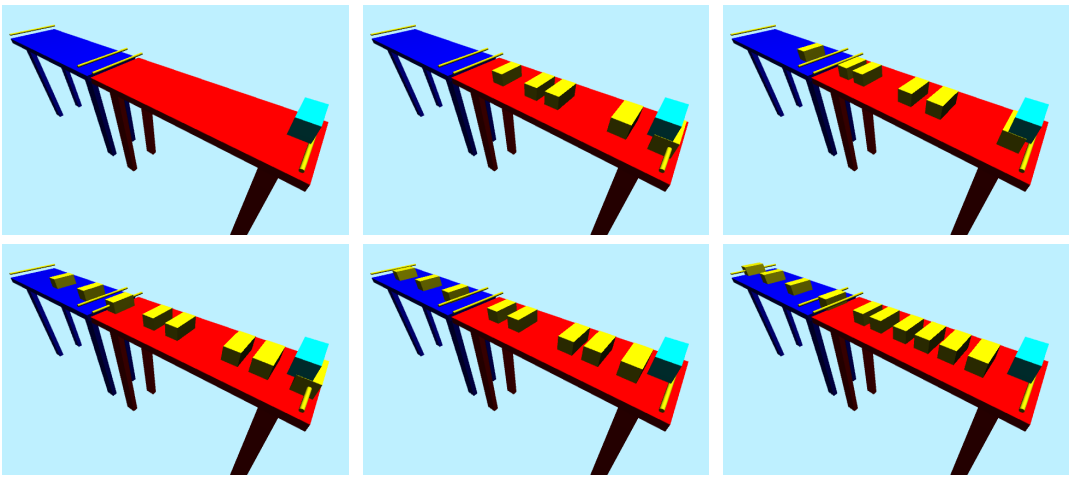


Figure 6.19: Several steps in the simulation of the belt conveyors

in the case of equal priorities we still need a resolution strategy. To make this work, the model has to be extended to support the annotation of priorities.

The solution we choose is to use priorities combined with nondeterminism in case of equal priority, as it can be applied without extending the types (and priorities were available from the mover bindings already). The main reason for this choice is that cases where such a resolution strategy is needed seem to be rare in typical automation systems, so the most convenient method was chosen.

6.7 Example: Belt Conveyors

To conclude this chapter we present a complete model featuring two belt conveyors with photo-electric barriers. The belts transport bricks and are controlled in a way to separate the bricks to a fixed distance. The example is simple compared to the models discussed in Chapter 8, but this allows us to describe it in full detail.

The setup is best explained by inspecting the simulation, which is shown in Figure 6.19 for some points in time. What can be observed is that the bricks are inserted (generated) at the start of the front (red) conveyor with random relative positions, while they are equidistant on the back (blue) conveyor. This is realized by detecting the bricks using the photo-electric barriers and stopping the belts accordingly.

The system is built from two conveyor components, which each consist of the belt and two photo-electric barriers at the front and back. Both conveyors are the same and only differ in their initial position (and color for illustrative purposes). A pure software component called *Controller* is used

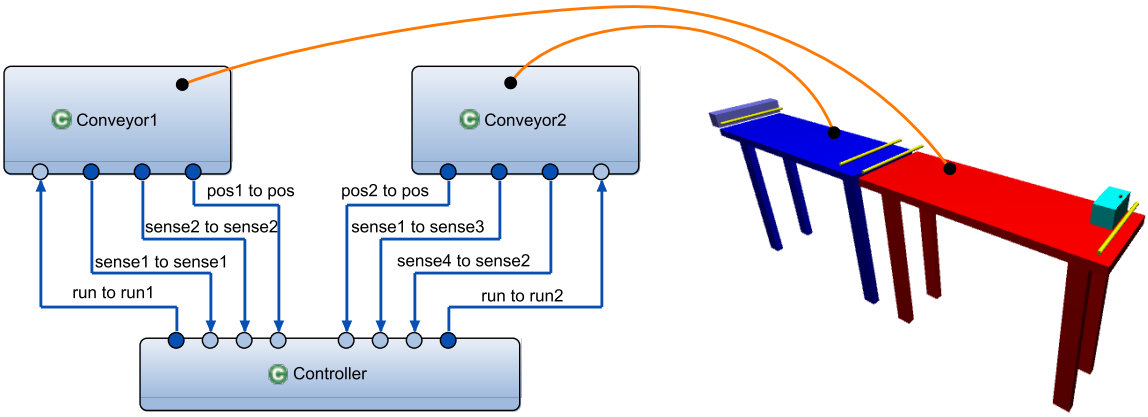


Figure 6.20: Top-level components and corresponding spatial elements

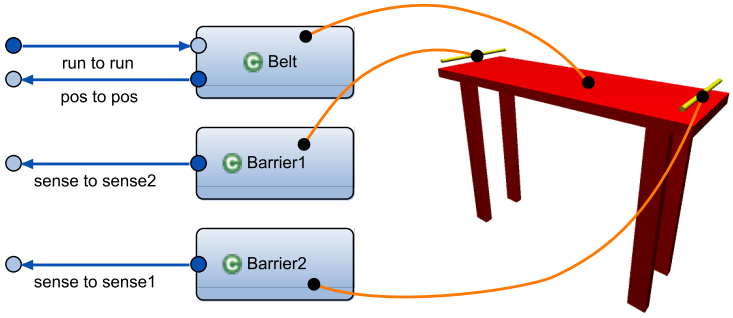


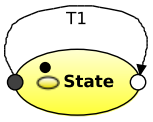
Figure 6.21: Conveyor component with corresponding spatial elements

to coordinate these conveyors. The entire setup is presented in Figure 6.20 and the details of the components will be explained next.

6.7.1 Conveyor Component

As the conveyor components are the same, we will describe only one of them here. It consists of three subcomponents, the *Belt*, which models the moving part of the conveyor, and *Barrier1* and *Barrier2*, which represent the photo-electric barriers. As can be seen in Figure 6.21, these components do not exchange data with each other, but are only composed spatially by fixing their relative position. The *Conveyor* component has four ports, where the input port *run* accepts a *boolean* indicating whether to run or stop the belt. Two of the outputs (*sense1* and *sense2*) indicate detection events by the two light barriers (type *boolean*), while the third, called *pos*, is of type *double* and sends the current position of the belt. The four feet of the conveyor are described by a single part of the *Conveyor* and are not part of any subcomponent.

comp Barrier	
outport:	sense : **boolean**
detector:	_ray



T1	**out:**	sense ! _ray

Figure 6.22: The component and automaton for the *Barrier* component

comp Belt	
inport:	run : **boolean**
outport:	pos : **double**
binding:	_binding
mover:	_mover: linear axis
	weakly bound by _binding

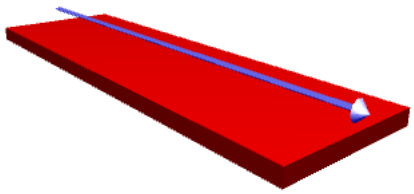
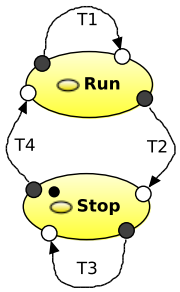


Figure 6.23: The syntactic interface of the *Belt* component

Barrier Component All light barriers used are modeled the same, so again only one of them is described here. Both the syntactic interface and the behavior defining automaton are shown in Figure 6.22. The volume of the detector *_ray* is the cylindric shape of the light ray. The automaton consists of a single transition that sends the current collision value via the output port.

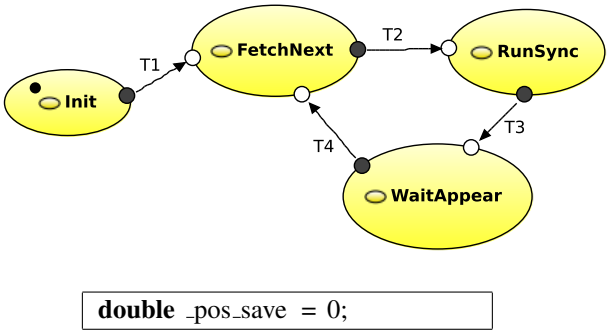
Belt Component The syntactic interface of the *Belt* component is described by Figure 6.23. The meaning of the ports has already been described in the context of the surrounding *Conveyor* component. The belt itself is modeled by a binding. Any material touching this volume is bound to the single mover which transports the material along the belt using the linear axis as indicated in the geometric view. The binding is chosen weak, as a blocked part will usually not cause the belt to stop but rather to slip through and thus the belt still affects other material placed on it. This allows the material to congest in front of blocking devices.



double _pos = 0;

Stop	**deq:**	_mover'=0; _pos'=0
Run	**deq:**	_mover'=0.1; _pos'=0.1
T1	**in:**	run?**true**
	out:	pos!_pos
T2	**in:**	run?**false**
	out:	pos!_pos
T3	**in:**	run?**false**
	out:	pos!_pos
T4	**in:**	run?**true**
	out:	pos!_pos

Figure 6.24: The automaton describing the Belt component's behavior



T1	**out:**	run1!**true**; run2!**false**
T2	**in:**	sense2?**true**
	out:	run1!**true**; run2!**true**
T3	**in:**	sense3?**true**; pos2?_X
	out:	run1!**false**; run2!**true**
	post:	_pos_save=_X
T4	**in:**	pos2?_X
	pre:	_X − _pos_save > 0.3
	out:	run1!**true**; run2!**false**

Figure 6.25: The automaton describing the *Controller* component's behavior

The behavior is again described by an automaton consisting of two states (*c.f.*, Figure 6.24) and a state variable *_pos*. The initial state *Stop* keeps both the mover and the state variable constant by using derivative 0[13], while state *Run* advances both at the constant rate of 0.1 units per second. All transitions send the current position value on the corresponding port and state changes occur based on the value received on the run port. The case of a missing input (ϵ) does not have to be handled, as the *run* port is modeled as a state port.

6.7.2 Controller Component

The component *Controller* has six input ports: *pos1* and *pos2* of type *double* are used to receive the position values from both conveyors, while *sense1*, *sense2*, *sense3*, and *sense4* are of type *boolean* and are connected to the four photo-electric barriers (via the conveyor components). Additionally, the two output ports *run1* and *run2* (type *boolean*) are used to switch each of the belts on and off. The behavior of the controller is defined as the single automaton shown in Figure 6.25. The *FetchNext* state waits for a brick to be detected at the end sensor of the first belt before switching the second belt on. In *RunSync* both belts are running, until the entry sensor of the second belt detects the brick (*sense3* port). When this happens, the first belt is stopped and the current position of the second belt is stored in a local variable. State *WaitAppear* then waits until the position of the second conveyor has progressed 0.3 units and then turns the first belt on again, while stopping the second belt. Obviously many of the ports are not used in the automaton. The reason to include them in the component's interface is that this model has been used to experiment with different controllers, from which some accessed more of the ports.

[13] Actually, these differential equations are not necessary in this case, as missing derivatives are interpreted to be zero. The equation is only given for instructional purposes.

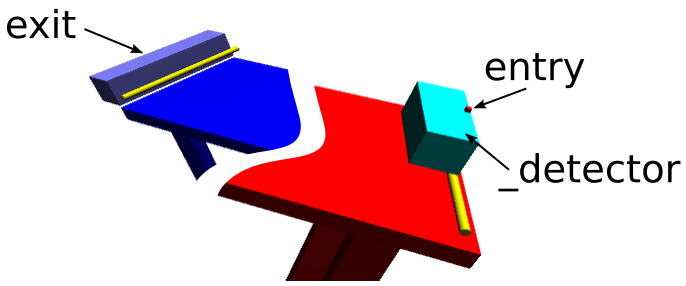
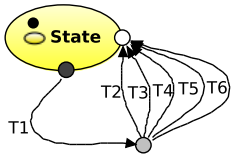


Figure 6.26: Detailed view of the material flow elements



double _v1 = 0.9;

State	deq:	_v1'=1
T1	pre:	_v1 >= 1 && ! _detector
	out:	entry_out !**true**
T2	post:	_v1=0
T3	post:	_v1=−1
T4	post:	_v1=−2
T5	post:	_v1=−3
T6	post:	_v1=−4

Figure 6.27: The automaton describing entry's generation sequence.

6.7.3 Material Generation

The component used to model material is called *Brick* and is so simple that no details need to be provided. It only consists of a single part that describes the shape of the brick. The shape can be seen in Figure 6.19, where each of the yellow cuboids on top of the belts is one instance of *Brick*. The behavior of the *Brick* component is empty, *i.e.*, defined by an automaton with a single state and no transitions. Of course the component can still be affected by the other components, such as the conveyors.

For this example the material flow specification is associated with the top level component in the hierarchy, as the material flow affects the entire system. It contains one entry located at the beginning of the first conveyor and one exit at the end of the second conveyor. The corresponding volumes are shown in Figure 6.26, where the entry is the very small red box, while the light blue larger box is a detector. The component type associated with the entry is *Brick*.

The semantics for the entry, *i.e.*, the sequence of bricks generated, is defined by the automaton shown in Figure 6.27. The automaton is rather simple and consists of a single state and one state variable _v1 which acts as a clock (thus the differential equation _v1'=1 is used in the state). The interesting transition is *T1*, which sends *true* on the port corresponding to the entry, which corresponds to the generation of a new brick at the location of the entry. This transition is only active if

both the clock has reached at least the value 1 and the area where the brick is about to appear is clear. The later part is checked by ! _detector , *i.e.*, the detector associated with the material flow defining component (as shown in Figure 6.26) reports no collisions. The remaining transitions leaving the local interface point[14] reset the clock to a value between 0 and -4, so between 1 and 5 second will pass between the generation of two bricks (or more if the place is blocked). The transition and thus the time used is chosen nondeterministically, so different input material streams are possible.

6.8 Summary

This chapter described an operationalization of the modeling theory from the previous chapter. The goal of the operationalization is to make the model applicable by defining a concrete meta-model and notation for the various aspects. Additionally, the various gaps (or variation points) of the theory were filled in our operationalization. The foundation for this is the fixing of a concrete type system and models for space and time. Based on this, several techniques for the description of a component's behavior function are provided. Such description techniques are required as it is usually not feasible to allow arbitrary mathematical functions in the application of a modeling technique or even a tool realization. The behavior description is based on communicating (pseudo-) hybrid automata with state variables, but alternative specifications based on geometric spline or table notations are sketched as well. These specifications can also be extended by orthogonal error mode descriptions. Finally, a concrete realization of the environment semantics (abstract physics model) is provided.

The operationalized model builds the foundation for the tool implementation presented in the next chapter. In fact the major part of the description given here was based on the implementation. Especially the meta-model is a subset of the implemented one and the examples of graphical syntax are just screenshots from the tool.

[14] According to the graphical syntax introduced in Section 6.3.2, the local interface point joins transitions. So, in a step always transition *T1* and one of the remaining ones will be taken (or none at all if the preconditions are not met).

7 Towards Tooling

To allow experimentation with our models for space-intensive mechatronic systems, we implemented a prototypical editor. Tooling is important, as modeling systems of realistic size is not feasible without them, and automatic testing and analysis techniques can hardly be applied to models written on paper. The tool is based on the operationalized model presented in Chapter 6. This chapter explains details of the tool realization, as well as the process activities supported by the model and tool.

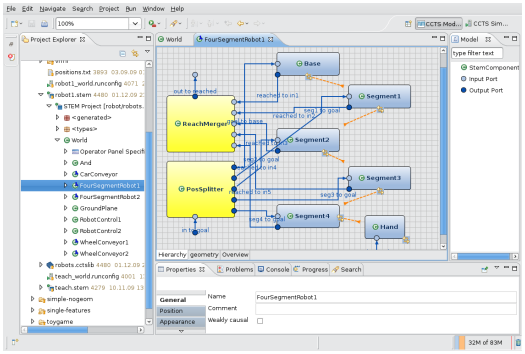
Section 7.1 provides a bird's eye view of the graphical editor and implementation details. The remaining sections deal with topics that are relevant for a practical application of the tool and thus the mechatronic behavior models, as well as activities which are enabled by representing the model in a machine readable form. In Section 7.2 we describe the library mechanism which allows systematic reuse of commonly used components. The link to the technical development models and a simple approach to tracing is summarized in Section 7.3, while Section 7.4 builds upon these technical links and explains how they can be applied for virtual commissioning.

7.1 Tool Overview

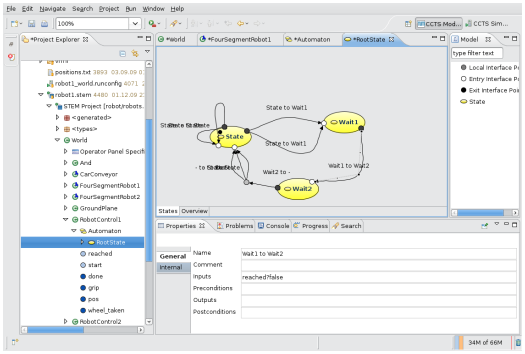
Our tool is a prototype of an engineering tool to support the development of space-intensive mechatronic systems. Its goal is to fill the gap identified in Section 2.1, where we explained that no suitable integrated behavior models (and tools) are available at the logical layer. This lack hampers fast exploration of the solution space and complicates the application of shorter iterations times during development. The prototype demonstrates how our approach could interact with other engineering activities, and of course is used to evaluate the model in various case studies (Chapter 8).

The tool prototype allows the creation, modification, and simulation of models that are consistent with our operational meta-model described in Chapter 6. Additional features provided are described in the other sections of this chapter. The editor, from which a screenshot can be seen in Figure 7.1(a), consists of a hierarchical view of the model on the left, specific property editors for the various elements of the meta-model on the bottom, and the editor area in the center of the tool. In this editor area more specific editors for different model elements can be shown. These more specific editors are:

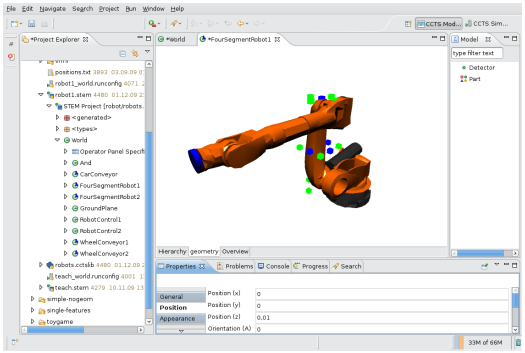
- editor for the type system (*c.f.*, Section 6.1.1);
- graphical editor for component diagrams (*c.f.*, Figure 7.1(a));



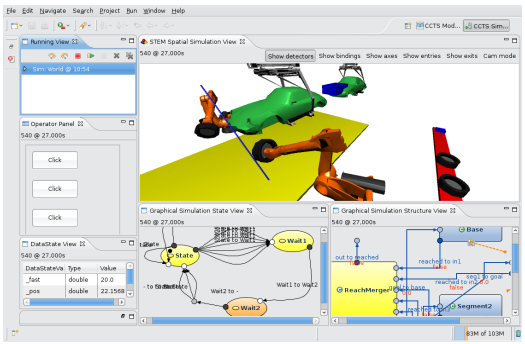
(a) Component diagram editor



(b) Automaton editor



(c) Geometry editor



(d) Simulator

Figure 7.1: Screenshots of the prototypical editor

- graphical editor for automata (*c.f.*, Figure 7.1(b));
- editor for graphical function specification (*c.f.*, Section 6.3.3, Figure 6.10);
- editor for tabular specifications (stream-based I/O tables, *c.f.*, Section 6.3.3, Figure 6.12);
- graphical editor for operator panels (*c.f.*, Section 6.3.3, Figure 6.13);
- graphical editor for error mode diagrams (*c.f.*, Section 6.3.4);
- editor for defining spatial aspects of a component, *i.e.*, parts, detectors, and movers (*c.f.*, Figure 7.1(c));
- editor for the technical resource model (*c.f.*, Section 7.3);
- editor for managing tracing links (*c.f.*, Section 7.3);
- editor for creating and modifying parameterized library elements (*c.f.*, Section 7.2).

Most of the editors are either structured editors based on table or tree representations of parts of the model, or graphical editors that are variations of the widely used box/circle and line diagrams. The exception is the editor used for the component's geometry, as it is based on a 3D-view of the model. The editor allows to create simple geometry based on parameterized primitives (boxes, spheres, cylinders); more complex shapes have to be prepared in external tools and can be imported using VRML[1]. All parts of the 3D model can be translated and rotated in our editor, so the composition of multiple imported VRML models with primitive geometry can be performed within the tool.

The simulator (*c.f.*, Figure 7.1(d)) is used to execute the models and supports the simulation both in real-time (*i.e.*, new simulation steps are triggered based on the time passed) and single-step mode. The later is essential for debugging the models and to better understand certain interactions between model elements. To further support the understanding of the models, the simulator provides various visualizations, which include a 3D view of the system's geometry, tabular views of the inputs, outputs, and states of the components, and a graphical display of component diagrams with the current messages exchanged, or automata with the current control state highlighted.

Technical Foundation The tool implementation is realized as a set of plug-ins for Eclipse[2], which is a Java framework for the development of rich client applications. Besides support of fundamental concepts, such as editors and views, and classes for constructing graphical user interfaces, it provides sub-frameworks for meta-modeling (EMF: Eclipse Modeling Framework) and the creation of graphical editors (GEF: Graphical Editing Framework). Eclipse also provides a plug-in concept with extension interfaces (called *extension points*).

The simulator has to deal with collision detection, as activation of detectors, evaluation of bindings, and congestion of parts are all defined via collision of their geometric shape. As the geometry model used is based on the three-dimensional Euklidean space, we can benefit from work done in the area of computer graphics and games in the last two decades. While there are still improvements in terms of algorithmic complexity for specific configurations, the general problem of detecting collisions of three-dimensional shapes constructed by primitive shapes and triangular meshes can be considered a solved problem. In our simulator implementation the collision detection library SOLID[3] is used [vdB97, vdB99] which is based on AABB trees[4]. The case studies (*c.f.*, Chapter 8) demonstrate, that on todays computing hardware the collision detector is fast enough to enable real-time simulation of complex models even for small simulation step sizes of about 10 milliseconds.

[1] The *Virtual Reality Modeling Language* is a description language for 3D scenes which was initially developed for usage in web sites. Many CAD modeling tools also support VRML as a exchange format for geometry.

[2] `http://www.eclipse.org/`

[3] `http://www.dtecta.com/`

[4] An AABB tree is a hierarchy of axis-aligned bounding boxes. These boxes are constructed to contain single spatial objects or sets of them. As these boxes can be checked efficiently for pair-wise collision, a lot of more expensive collision tests on general three-dimensional objects can be skipped based on the information from the AABB tree.

	Plug-Ins	Lines of Code	Classes in Meta-Model	Lines of Generated Code
CCTS	13	57,000	176	105,000
Prototype	10	42,000	93	74,000

Table 7.1: Key metrics for the size of the CCTS library and our tool prototype.

Conceptional Foundation The conceptional foundation of the tool implementation is partially provided by the AutoFOCUS prototype [BHS99, SPHP02], and especially its third invocation called AutoFOCUS 3. While AutoFOCUS 3 has been developed in parallel to this thesis from a different team, both AutoFOCUS 3 and our tool are based on a common library (called CCTS) which has been developed by the AutoFOCUS 3 team in collaboration with the author of this thesis. This library provides the type system (as described in Section 6.1.1) and basic versions of the meta-models for components, automata, and operator panels. Additionally, graphical editor support for these parts of the meta-model is provided, as well as supporting classes for model simulation. An important property of this base library is its extensibility, which allows both the extension of the meta-model and the modification and enhancement of properties of the tooling platform via well-defined interfaces. The main addition of our tool to the base library are the modified meta-model, changes to the provided editors (*e.g.*, to support differential equations in the automaton editor), support for modeling material, and description support for the spatial aspects of the system in terms of a simple 3D editor. To give an impression of the size of the CCTS library and our addition to it, some key metrics are summarized in Table 7.1.

7.2 Structured Reuse and Parameterization

Components in industrial automation systems are usually used multiple times, such as drives, belt conveyors, or entire industrial robots. This redundancy in the real-world system also leads to identical components in the model. The occurrence of identical parts in a model is even beneficial, as it allows the reuse of existing (sub) models and the construction of a library of commonly used and well tested sub-models which can be easily integrated into the model of a new machine. This reuse can significantly speed up the construction of new complex models, as only the remaining parts which are not available in a library have to be modeled (*c.f.*, [JGJ97]).

The easiest method for reusing parts of existing models is white-box reuse by applying copy&paste. However, copy&paste has some major drawbacks known from code-based development: (1) the induced redundancy increases the maintenance efforts [Kos07, RC07], especially changes to the duplicated parts have to be repeated for all copies, (2) the presence of duplicated parts often lead to inconsistent changes, which in turn often cause incorrect behavior [JDHW09, BFG07], and (3) for certain reuse scenarios simple copy&paste is just not sufficient as too many changes would have to be performed to the duplicated part (an example for this is given in Section 8.2). While cloning

in source code is a much studied subject, recent work indicates that cloning is also much used in model-based development [DHJ+08, DHJ+10].

To provide a reuse mechanism without the problems of simple copy&paste, we implemented generative libraries as described in [HH09] for our tool. In the generative library approach arbitrary model elements may be used as library elements by organizing them in a special container hierarchy, called the *library*. Any element in the library can be inserted into the model, which causes a copy of the library element to be created. The new element (called *instance*) is marked by a *LibraryReference* model element, which points to the library element it was created from. While this is similar to copy&paste at a first glance, the additional reference allows instances to be updated when the library element is changed. In addition, the editor must ensure that instances of a library element are not modified (read-only parts of the model), which enforces that all changes in reused elements are performed in the library element. More of the technical details are provided in [HH09].

The generative libraries avoid the drawbacks of copy&paste, as an explicit link between the original and its duplicate is preserved. In contrast to a heavy-weight library mechanism that is realized in the meta-model (*i.e.*, with explicit instance classes), the changes to both the meta-model and the existing tooling infrastructure (editors, simulators, code generators) are minimal, as the internal representation of the model explicitly contains the duplicated parts.

Our implementation also allows the creation of parameterizable library elements, so for example a library element of a belt conveyor could support a variable user provided length. For this, an instance of a library element can be annotated with additional parameters, which are used as input of a model transformator which modifies the copied sub model after the copy operation and before being pasted. Both the possible parameters and the transformation are a part of the library element's definition. Besides mere parameterization, this mechanism allows a generative construction of complex parts of the model. An example is the transportation chain described in Section 8.2, to which more than 100 tool carriers are attached. Instead of modeling and positioning all carriers manually, only one of them is described and then duplicated and positioned by the model transformation during the creation of the instance. This way the number of carriers can also be provided as a parameter, which allows a simple way to switch to a less complex version of the model with less carriers during debugging.

7.3 Link to Technical Models

Typically a logical behavior model is not an end in itself, but rather will be one artifact in a larger development or maintenance process. As outlined in Section 2.1, the model described in this thesis would be used at the logical layer which describes and explains the inner working and logical solution of an engineering problem. It acts as a bridge between the functional or usage layer, which captures *what* the system should do, but not *how* it is done, and the technical layer, which describes the resources used to realize the logical behavior. In the factory automation domain the technical

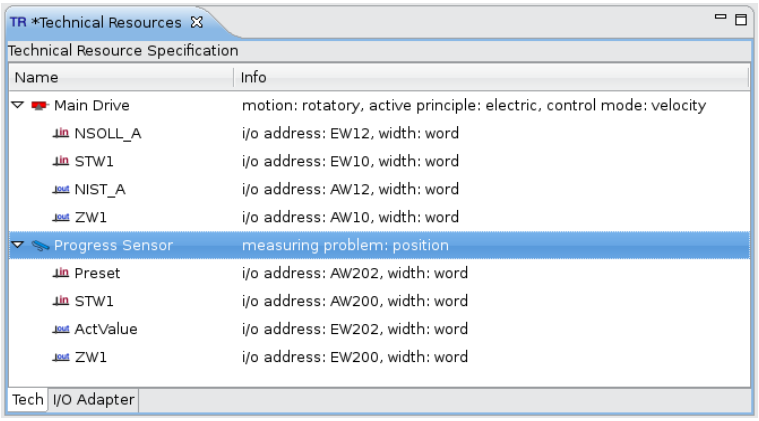


Figure 7.2: The technical resource model for the Belt Component from Page 119.

layer is typically provided by CAD models for mechanics, fluidics, and electrics, as well as PLC code and hardware configurations for the software parts.

To be useful in the larger development process, the logical model must somehow support the transition to the technical layer. One way to achieve this is to gradually enrich the logical model by more technical information about the mechanical and electrical parts, such as which actuators and sensors to use and the engine power required. This information together with the component structure can then later form the foundation for creating the more detailed technical engineering models. To support the annotation of such technical information in the editor, we allow components to be augmented by what we call the *technical resource model (TRM)* [BHLH09]. It captures the technical details of a component's realization. In particular, it stores descriptions (*e.g.*, vendor, model number, principle of operation, *etc.*) of sensors and actuators utilized by a component. The benefit of managing the TRM in the same model is that the logical model and the information from the TRMs are more easily kept consistent. An example TRM for the *Belt* component from the example in Section 6.7 can be found in Figure 7.2. In the TRM the main drive which is responsible for moving the belt is modeled as an actuator, while the position value is realized by a respective sensor. For both the technical/binary interface is included as well, which is used when generating simulation models for virtual commissioning.

All of the information in the TRM is of purely syntactical nature with respect to the behavior model, *i.e.*, it does not influence the behavior of a component. However, the additional information allows to collect technical requirements early on in the design process and augments the logic model during the creation of the models for the technical layer. The TRM also forms the foundation for the automatic generation of simulation models for virtual commissioning (see Section 7.4).

Tracing and Consistency When working with different models which are related to each other, such as the models from the logical and technical layer, it is important to keep these models

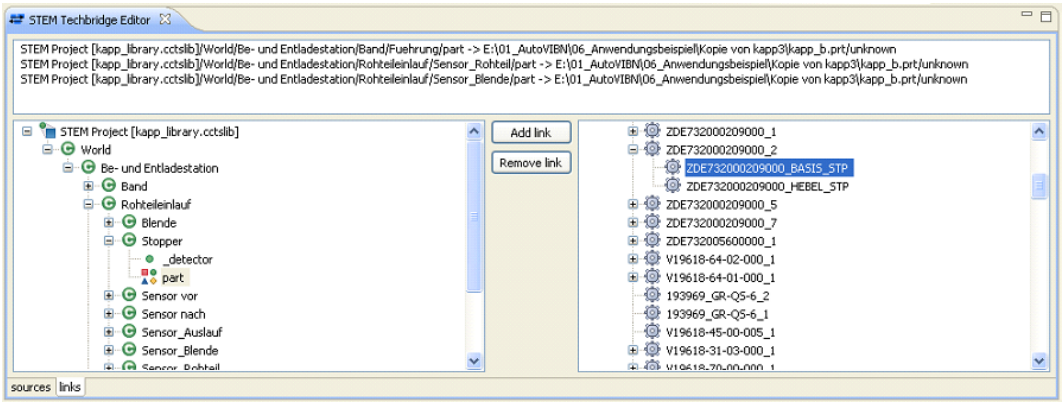


Figure 7.3: The editor used to link model elements in the tracing file.

from diverging. For one-time and strictly top-down development the consistency of these models does not seem to be relevant, as returning to the logical layer, once the technical layer has been completed, is not required. In a realistic context, however, there will always be iterations in the development process, either planned as part of an iterative or incremental development process, or unplanned in the case of problems encountered on the technical level which require a partial redesign on the logical level. Additionally, keeping all created models up to date even after the construction and assembly of an automation system can be advantageous when reusing parts of the models for new developments and paves the way for post-development activities, such as model-based diagnosis and maintenance, or training of machine operators.

To solve this problem, we follow the common approach of managing explicit links between the model elements of the various models. For this mapping the TRM is relevant, as is allows the elements from the technical models to be connected directly to an actuator or sensor (and thus indirectly to a component). The logical behavior model acts as the central hub, to which models from the technical model are related to. These links are managed in external tracing files, which map elements to each other by using internal element IDs, which are managed by all involved modeling tools. Creation and management of these links is related to the user, who has to connect corresponding elements using the graphical editor shown in Figure 7.3. If there is a canonical mapping between model elements, be it based on congruent hierarchies or naming conventions, the approach can easily be extended to also include this mapping to release the engineer from a part of this work. In the case of forward engineering (*i.e.*, partial generation of the models in the technical layer), the corresponding tracing files can be created automatically as a side-product.

The main purpose of the tracing links besides documentation purposes and rationale management is consistency checking. During consistency checking, linked elements and their properties are compared with each other to find deviations and also identify elements which are not covered by the tracing links. These checks can for example compare the types of actuator used (which can be found in the TRM), or the position of a part in the 3D-CAD model with the one used in the integrated

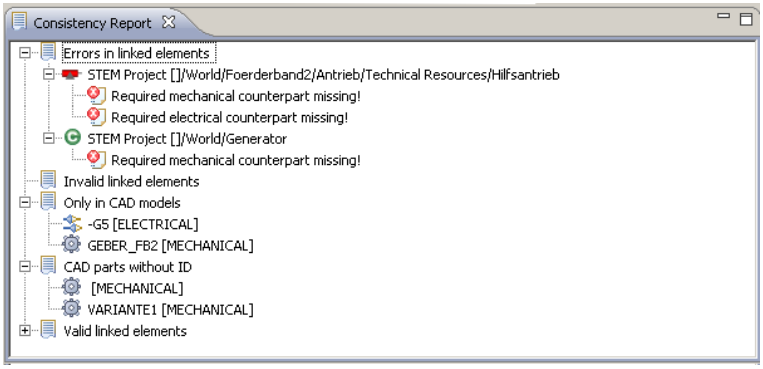


Figure 7.4: Example of a consistency report.

model. This way the divergence of the used models can be detected and avoided early, keeping all models meaningful during the entire development phase. An example of such a consistency report is given in Figure 7.4 and consists of the following sections:

- *incomplete links*, *e.g.*, if the electric counterpart is provided but not the mechanic one,
- *invalid links*, *e.g.*, linking an actuator in the model to an device in the electric CAD that corresponds to a sensor,
- *unlinked CAD elements* without a counterpart in the logical model and hint at missing tracing links,
- *CAD elements without ID*, which hint at errors during data extraction from the CAD tools, and
- *valid links*, which are listed for random manual inspection of tracing links.

7.4 Virtual Commissioning

Virtual commissioning is the process of testing the controller software of a mechatronic system with a virtual machine model[5]. As argued in Section 1.2.2, the controller software and its input and output signals are hard to interpret without the actual machine being controlled. Consequently, the testing of complex controller software is practically infeasible without the machine, especially as preconditions (*i.e.*, valid inputs) are often implicitly encoded in the physics of the system. To avoid testing at the real machine, which can only be performed after assembly and might damage the machine, often virtual commissioning is applied. This allows the software tests to happen earlier

[5]The term virtual commissioning is used predominantly in the automation domain. Other domains use this technique as well, but often use a different term for it.

in the development process, which can help to reduce the overall development time and can have a positive impact on software quality due to reduced time pressure (*c.f.*, [ZWHL06]).

The typical setup for virtual commissioning is to run the controller software on the real controller hardware, which is then connected via a field bus and an interface board with a computer running the machine simulation. This setup is also referred to by the term hardware-in-the-loop or HIL testing, as it involves the actual controller hardware, while software-in-the-loop or SIL testing indicates that the controller software is running in a virtual controller as well. In the latter case, both simulations can run on the same computer and thus interfacing via a field bus can be omitted. In both cases the actual PLC software is executed, not just an abstract model of the software.

The creation of virtual commissioning models is a time-consuming and error-prone task, since information needed is spread over different artifacts of the engineering process. This information is manually gathered and incorporated into the simulation models, producing redundancy and causing additional maintenance effort. In particular, for these virtual machine models one needs geometry from mechanical CAD, wiring plans and bus topologies from electric CAD, signal-level communication protocols from PLC projects, as well as the rather implicitly available knowledge about the behavior of the used mechatronic components, material, and its flow through the system.

As the relevant information is already present in the logical behavior model, it is an obvious step to reuse this information for the creation of the virtual machine model. To allow the extraction of such a model, the behavior model must be detailed enough to allow all primitive components to be clearly classified as either a part of the software or the electro-mechanical parts of the system. If this condition is met, the virtual commissioning model can be created by removing all software related components from the model. However, while the behavior model communicates via logical signals, the controller software is built to work with the real hardware and thus uses bit-level messages on the field bus level. For example, a drive in the logical model might receive a simple on/off signal, while the controller software sends a 16 bit number where every bit has a special meaning, such as *quick break* or *release*.

To bridge this gap, a mapping between the logical signals and the actual hardware messages has to be established, which can be interpreted as a simple form of interface refinement. The additional information required for this is located in the TRM introduced in the previous section. As can be seen in Figure 7.2, each actor and sensor in the TRM is also annotated with the hardware signals used (typically these are either individual bits or entire 8 or 16 bit control words). These signals contain both a logical name and a hardware bus address. The actual relation between port messages and hardware signals is described in the *I/O mapping*, which provides for outgoing ports (from the machine view) the translation from logical messages to bit patterns on the signals, and for incoming ports the mapping from bit patterns to logical messages. We assume that the mapping between logical and binary messages is independent of time and state, and thus simple functions are sufficient. In the seldom cases, where this assumption does not hold, the corresponding translation logic must be moved into the logical model, where more complex relations can be captured using automata.

The tool implementation allows the automatic generation of the complete virtual commissioning model from the logical behavior model, once the interfaces are fully described in the TRMs. The model is generated as C++ code, which can be compiled to a program that simulates the machine and directly communicates with a PROFIBUS[6] interface board, thus allowing interaction with the controller hardware. The simulator of our tool can be connected to this program via TCP/IP. This allows to apply the same visualizations used during *normal* simulation to the internal state of the machine simulation. Additionally, it supports limited interaction, such as pressing (virtual) buttons in operator panels. As the virtual commissioning model is fully generated (and should not be changed manually) it can be regenerated after changes to the behavior model and thus always stays consistent with it. As the consistency of the behavior model with the technical (CAD) models can be automatically checked (described in the previous section), we indirectly also ensure consistency between the technical models and the virtual commissioning model, which is a problem in many existing approaches.

7.5 Summary

This chapter provided an overview of the tool implementation of our modeling theory. The tool is the foundation for the case studies presented in the next chapter. In addition, working on the implementation of the tool and with the tool itself helped to improve the theoretical model. The tool also demonstrates possible process support for systems engineering, such as tracing and virtual commissioning. While the tool implementation is just a prototype which is certainly not fit for application within the time and resource pressure of real development projects, the availability of the tool greatly helped in the discussion with domain experts. Bringing the theoretical model to life by simulation and automated processing, helped to get the underlying ideas communicated to both colleagues from the mechanical engineering department and industrial partners.

[6] PROFIBUS (Process Field Bus) is a field bus that is commonly used for local communication in automation systems [IEC07].

8 Case Studies

This chapter summarizes three case studies that have been performed to evaluate our model. The goal of these studies is to explore whether real systems from the factory automation domain can be expressed using the model from the previous chapters. We describe both the original systems, the most interesting parts of the models created, and lessons learned from modeling itself.

In the following section, the case study design is presented. After this, we provide details on the systems and models used in the three case studies. The first two case studies are parts of existing machines from machine tool vendors. Their models have been created in the context of the BMWi[1] funded project AutoVIBN [BHH+10] and have also been used for virtual commissioning. The third study features industrial robots, but has no real-world counterpart. It has been created to have an example with deep kinematic chains and complex motion paths in the material flow. We conclude this chapter by summarizing the results from the different examples and studies.

8.1 Case Study Design

The focus of the case studies presented here is the model itself. We intend to evaluate both the expressiveness of the model and the effort required to build a model of a system. While an ideal case study would evaluate the benefits of our model and approach for the system development process, this is nearly infeasible in practice. A typical setup for this would be two different teams that develop the same system, one using the *traditional* approach, one using ours. While experiments targeted in this direction can yield interesting insights, there are various obstacles in practice. One is the large number of variables which can not be controlled very well. For example, differences in performance between both teams could be due to their previous experience and not the overall approach actually used. In addition, to make the results transferable, the system being developed has to be non-trivial and of realistic size. Performing such an experiment in an industrial setting using novel processes and tools which are not yet mature is usually to much of a risk for a company, so finding suitable projects is nearly impossible.

To better understand the expressiveness and appropriateness of our model, we used (parts of) existing industrial systems[2] and modeled them using the tool presented in the previous chapter. The input typically consisted of existing CAD models created during the development of the systems

[1] Bundesministerium für Wirtschaft – German Ministry of Economy

[2] For the third case study only the robot itself corresponds to an existing system, while the overall model is designed similar to typical applications of industrial robots.

and initial discussions with engineers of the industrial partner. During modeling we recorded how well the individual parts could be described by the model. Where possible, we also discussed the resulting model with the engineers responsible for the system. The required modeling effort could not be determined exactly, as the work had to be suspended frequently, either due to bugs discovered in the modeling tool or for clarification questions to the developers of the original system. However, qualitative effort estimates were collected.

One important variable in a modeling case study is the *person* using the tool. As outlined in Section 2 the intended usage of the model is in the early development stages where different alternative solution strategies might be explored and the logical processing steps performed by the system are fixed. We envision this modeling to be performed by a *mechatronic design team*, which has members of all required engineering disciplines. The model then serves both as input to discussions within this team to better explain design ideas, as a catalyst during discussions to exchange and try solution alternatives, and also as the output of discussions to document the agreed solution. The different members of the team represent the discipline specific views on the system and are responsible for the corresponding parts of the model.

Unfortunately, we did not have access to such a mechatronic design team from an industrial partner. One problem is that many companies do not yet have such teams that are cross-cutting through the departments. The second obstacle is that a certain amount of training with both the model and the tool is required to be actually useful. So, the company would have to spend more resources than just the modeling time. To deal with this situation, we collaborated with researchers from a mechanical engineering chair at TUM[3]. The model for the first case study was modeled completely by a colleague from this chair with only little support from the author. The second case study was modeled collaboratively by this mechanical engineer and the author, where the engineer mostly worked on the geometrical parts including sensors and actors, while the author concentrated on the control logic. Thus, this setup somewhat resembles the mechatronic design team. The third case study was modeled by the author alone.

The next sections each introduce the systems and models of each case study. Results specific to each case study are discussed there, while the overall results are collected at the end of the chapter.

8.2 Virtual Commissioning at Heller: Tool Magazine

Gebr. Heller Maschinenfabrik GmbH is a machine tool vendor located in Nürtingen in southern Germany. It was founded in 1894 and today operates world wide. Its main business is the development and construction of machining centers and automation solutions. The system being modeled is a part of a multi-axis machining center similar to the one shown in Figure 8.1(a). The right part of the center contains the workspace of the machine where a milling or grinding tool is applied to a workpiece based on paths derived from CAD models. The number of axes available determines

[3] Institute for Machine Tools and Industrial Management

(a) The entire machine.

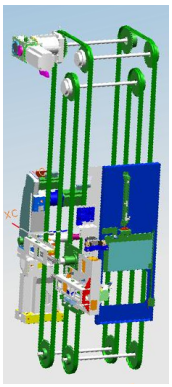
(b) View of the tool changing door.

Figure 8.1: Two views of a Heller multi-axis machining center.
Images are courtesy of Heller GmbH.

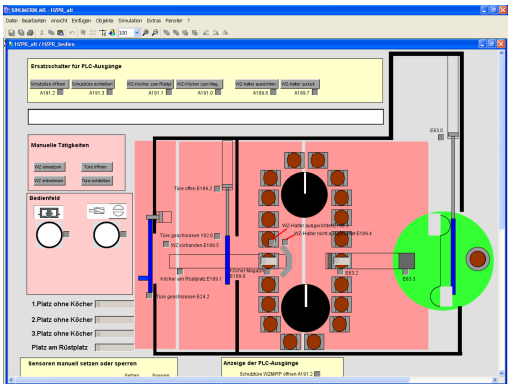
the complexity of the parts that can be processed. The left part, which is slightly higher than the rest of the machine, contains the tool magazine. For the different process steps applied to a single workpiece often different tools are needed. Additionally, the tools wear off during usage and have to be replaced with new ones after a certain time of use. To minimize the need for time-consuming manual intervention, up to two hundred tools can be stored in the tool magazine and changed autonomously by the machine. As tools wear out, they still have to be replaced from time to time. To keep the machine from being idle, this can be performed in a process called *main-time parallel setup*[4] where tools can be inserted into or removed from the magazine while the machine processes a workpiece. The door which is used for this tool exchange can be seen on the very left of Figure 8.1(a) in front of the operator. Figure 8.1(b) gives a better view of this door, where also the carriers for the tools are visible (the black parts in the back). The part of the machine modeled is the tool magazine including the safety door used to exchange the tools. The goal was the creation of a model suitable for virtual commissioning. Thus, the focus during modeling was less on the software aspects, but rather on the hardware parts.

Modeled System The input for modeling the system was provided by a mechanic and an electric CAD model (*c.f.*, Figure 8.2(a)) of the relevant section of the machine. These models were available as the construction of the system already was completed at that time. Additionally, a Heller engineer provided an overview of the modeled part in a short workshop. Earlier attempts at virtual commissioning at Heller resulted in a simulation model described with the tool *Sinumerik Machine*

[4]German: Hauptzeitparalleles Rüsten



(a) Mechanic CAD model



(b) Simulation model for virtual commissioning

Figure 8.2: Existing models for the system

Simulator (Figure 8.2(b)). This model was analyzed as well for understanding some details of the internal processes.

Model Overview Our behavior model of the system consists of 517 components (including those used for assumptions and guarantees of the 161 *MaterialFlowSpecs*), 501 automaton specifications with an overall of 669 states (so most automata are fairly simple), and 188 TRMs attached to components. The geometry of the model is depicted in Figure 8.3. The most prominent elements are the chain of the tool magazine and the operator, whose position is indicated by some geometric elements on the right-hand side. In the zoomed-in view from Figure 8.4 also the safety door (blue), the tool extractor (yellow), and the alignment unit (green) can be seen. Additionally, there are various light barriers which are used to check the current position of the safety door and the tool extractor, and also detect whether a tool is present in the tool extractor. The meander shaped chain (*c.f.*, Figure 8.5(c)) contains 160 clip positions which can each hold a single tool. The chain can be freely moved forwards and backwards, thus allowing each position of the chain to be located at the operation entry.

There are three different kinds of material, two are the red and light-blue tools, which can be seen in the pictures. The third one is the *tool carrier* (green), which is attached to each tool and acts as the connective element between the tool and the chain. For the material flow, each of the clip positions in the chain supports the generation of material (both the tool carrier and the tool) as an easy way to initially fill the magazine. Additionally, the operator is associated with a material flow specification, which allows to remove a tool and insert a new one. For the new tool the light-blue one is used to allow easier recognition in the simulation.

As the model is just used for generating the simulation for virtual commissioning, only a very basic controller software is included. It supports the activation and movement of all elements of the

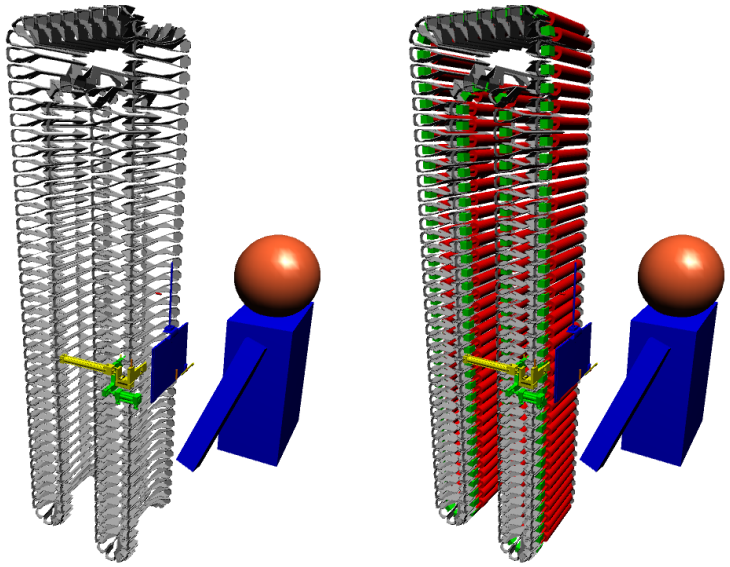


Figure 8.3: Complete view of the geometry of the tool magazine without and with material.

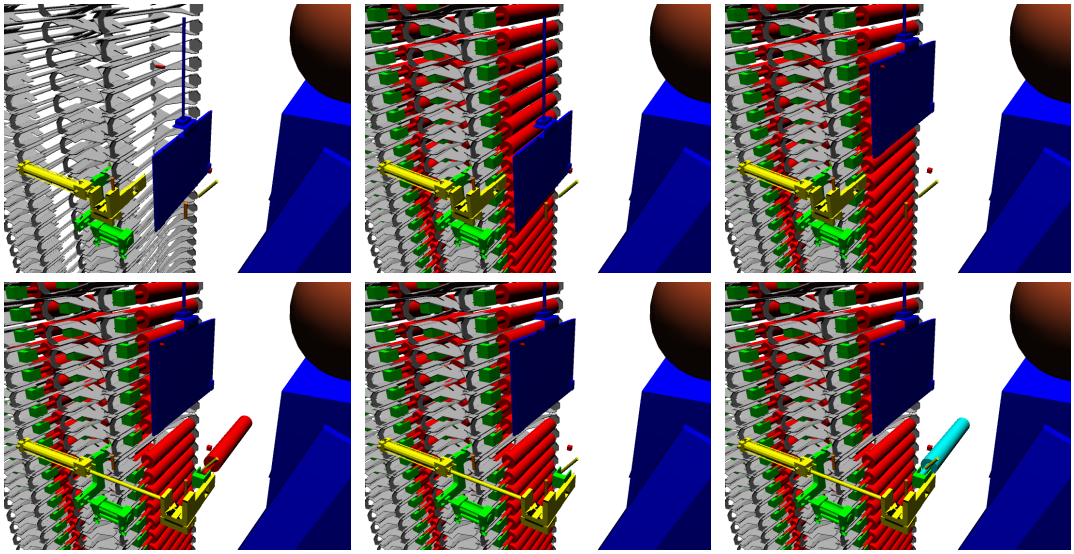
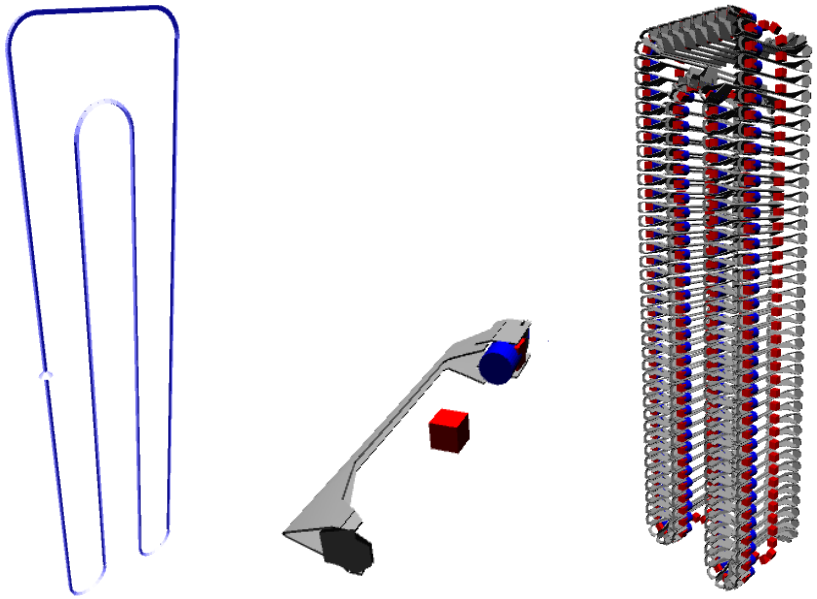


Figure 8.4: Detailed view of the steps involved in the tool change process. From left to right: (1) system without material, (2) simulation initialized with material, (3) safety door opened, (4) tool extracted, (5) tool removed by the operator, (6) new (light-blue) tool inserted.



(a) The curve for the chain (b) A single chain element (c) The assembled chain

Figure 8.5: The ingredients for modeling the chain

system via an operator panel, including the opening of the safety door, and moving the chain to a certain position (indicated by the index of the clip position). All non-software components are annotated with TRMs including the mapping to hardware signals, which then allows the automatic generation of the entire virtual commissioning model.

Model Details One of the challenges for this model was the chain in the tool magazine with its 160 clip positions and attached tools. Even with copy&paste this is tedious and error-prone to model manually. It can be argued that this level of detail is not required for a purely logical model, and the existing virtual commissioning model (*c.f.*, Figure 8.2(b)) only managed the contents of the chain using a counter for the number of contained tools. However, there are situations where the full detail of all 160 chain elements including their geometry is required, *e.g.*, when simulation should ensure that no unexpected collisions occur. Additionally, we decided to represent the full chain geometry to evaluate the feasibility of doing so.

The meander shape of the chain was extracted from the existing CAD model and imported into the tool as an interpolated axis by pasting coordinates of interpolation points (Figure 8.5(a)). Next, a single element of the chain was modeled as shown in Figure 8.5(b). The gray base geometry was extracted from the CAD model and imported as VRML. The red and blue parts are the positions used to generate new material (entries of the material flow specification) and the binding used for moving clipped in tools with the element. With these two elements the library mechanism described

in Section 7.2 could be used with a small script to create the complete chain from Figure 8.5(c). This way the entire chain can be constructed within less than one hour and easily adjusted later on. For example, the positions of the entries had to be corrected after initial simulations. Being a library part, only the single chain element had to be changed to affect all 160 elements of the chain. Additionally, as the number of chain elements was made a parameter of the chain library element, the number of chain elements could easily be reduced to any number during testing of the model (and debugging of our tool).

The second major challenge was the multi-level material flow, as the tools are not directly transported, but rather attached to a tool carrier. As both the tool and the tool carrier can be moved freely and may exist independently (during tool extraction, the tool is removed from the simulation, while the carrier remains), both had to be moved as individual material objects. The tool carrier was modeled with a binding that causes any tool attached to it (*i.e.*, colliding with the binding) to follow its motion.

Results All elements of the chosen part of the tool magazine could be captured using our tool and thus the underlying model. The size and complexity of the chain demonstrates the importance of a library and generation mechanism for repetitive parts. It also shows that simulation of such a model can be performed fast enough, as the model runs in real-time on current computing hardware. The model was used to generate a machine model for virtual commissioning, which was successfully executed in conjunction with the real controller software (the setup is shown in Figure 8.6). As not the entire machine but only a part was simulated, some minor adjustments had to be made to the software to not rely on the non-existing parts. As Heller had already used this software for virtual commissioning before, it was already prepared for this.

One lesson learned during the virtual commissioning, besides feasibility, is that many properties of the system are nowhere documented. For example, the speed of the door (or at least the pneumatic cylinder operating it) was not included in any documents. The software, however, contained a timeout and went to an error state as our machine model did not send the *open* signal fast enough. The door was too slow in our model. After manually extracting those implicit constraints from the code and adjusting our model, the simulation worked well together with the controller and allowed to perform the full tool change cycle.

8.3 Virtual Commissioning at Kapp: Loading Mechanism

Kapp GmbH constructs machine tools for gear grinding and is located in Coburg. It was founded in 1953 and since 1997 the Niles GmbH is a part of the Kapp Group. Their machines are mainly used to give the finishing polish to high precision gears with diameters up to 4.5 meters. Such gears with tolerances below a micrometer are typically used in the automotive and avionics domains, while the larger ones are used in wind generators. In a typical customer relationship the machine

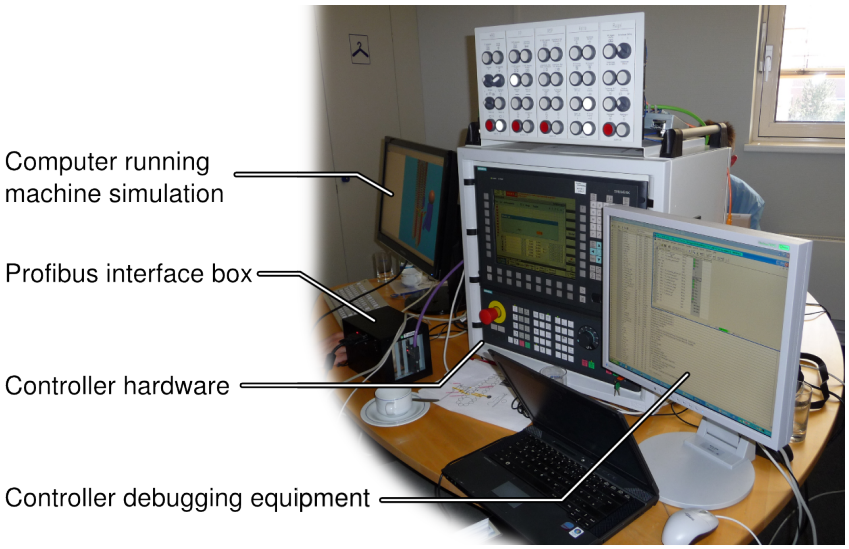


Figure 8.6: The setup used for virtual commissioning at Heller

tool itself is mostly adapted via well-known parameters of the system. Contrary, the integration with the assembly lines of the customer typically requires a custom solution and thus is developed newly, possibly based on experience from earlier systems. As in such a setting the latter part would benefit the most from a logical behavior model, a loading/unloading mechanism of a gear grinder was chosen as being modeled.

The goals of this case study were twofold. First, as with the Heller case study, a model suitable for virtual commissioning should be created. Second, the modeling of reusable mechatronic components should be evaluated.

Modeled System A 3D view of the loading system is shown in Figure 8.7. It consists of a circular transportation belt, which turns in clock-wise direction (when seen from the top). The right area is accessible to the operator, while the left side is usually protected by a fence. The gears are mounted on carriers (small pallets), which are transported by the belt. The operator is responsible for placing unprocessed gears onto the carriers and extracting them once they have been processed by the machine. As the processing time is in the order of several minutes and multiple gears can be inserted and extracted due to the length of the belt, this manual interaction with the system only has to happen once every couple of hours. Besides transportation, the main task of the loading system is to ensure that only gears of the correct size and only unprocessed gears enter the machine, as in both cases the gear could be damaged beyond repair by the grinder. In addition, there are a couple of stopping positions which are monitored by the system. After entering the closed part of the loading system, a carrier will first reach the hand-over point to the machine, where it is lifted and taken over by a gripper, which belongs to the grinding machine and thus is not a part of the model. The next

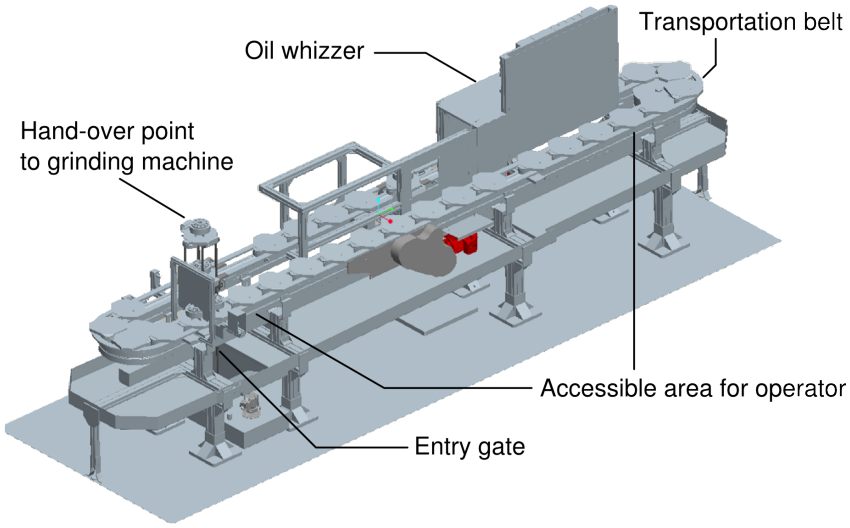


Figure 8.7: Overview of the loading/unloading system (CAD model courtesy of Kapp GmbH).

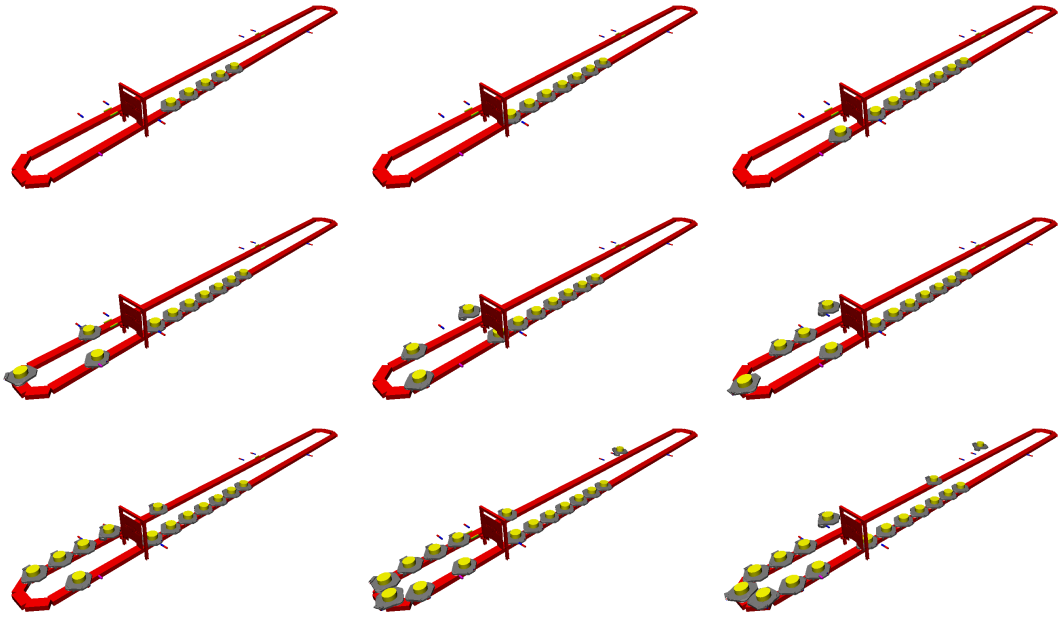


Figure 8.8: The model of the loading/unloading system in action.
Closer views are provided by Figures 8.15 and 8.16(b).

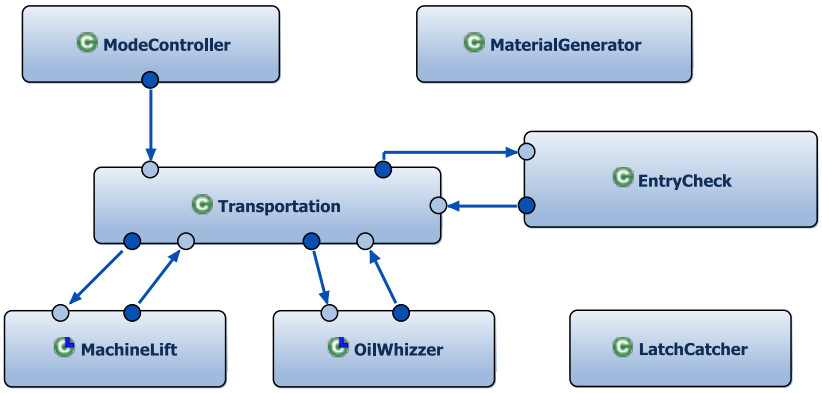


Figure 8.9: Top-level structure of the Kapp model.

stop is an oil whizzer that is used to remove most of the oil, which was applied during grinding, from the gear. After this step the gear proceeds to the exit of the closed area and becomes accessible for the operator. Several steps of a simulation of the model are shown in Figure 8.8. The machine and the oil whizzer themselves are not modeled, but only the lifts at the corresponding hand-over points.

Model Overview As one goal for this case study was to demonstrate mechatronic modeling (in the meaning of tight coupling of mechanics and software), the structure of the model is explained in more detail here. The model consists of 75 components, whose behavior is described by 10 operator panels and 50 automata with a total of 150 atomic states. Its overall structure can be seen in Figure 8.9. The central part is the *Transportation* component, which exchanges messages with the *MachineLift* and the *OilWhizzer*, which represent the corresponding hand-over points, and the *EntryCheck*, which corresponds to the entry gate. The *ModeController* encapsulates a simple button, which can switch between *manual* mode, where every part of the system can be controlled by the operator, and *automatic* mode, where the parts are controlled by the machines programmed logic. The *MaterialGenerator* is not a part of the machine, but rather represents the operator. During simulation, this component can be used to insert carriers and gears into the model. Finally, the *LatchCatcher* is a very simple purely mechanical part and thus has no communication connection to the remaining machine. The purpose of this part will be explained later on.

It is already visible from the top-level structure, that the *Transportation* component is pivotal for the operation of the loading system. This component is responsible for transporting the carriers to the individual stops and dealing with congestion. Figure 8.10 depicts a bird's eye view of the transportation belt and associated sensors and stoppers. The yellow cylinders are proximity sensors, while the red/blue cylinders are stoppers. The sensors and stoppers belong to four logical groups, which are labelled in the figure. This structure is also reflected in the component hierarchy, shown in Figure 8.11. Each of the four groups consists of one or two stoppers and a couple of sensors.

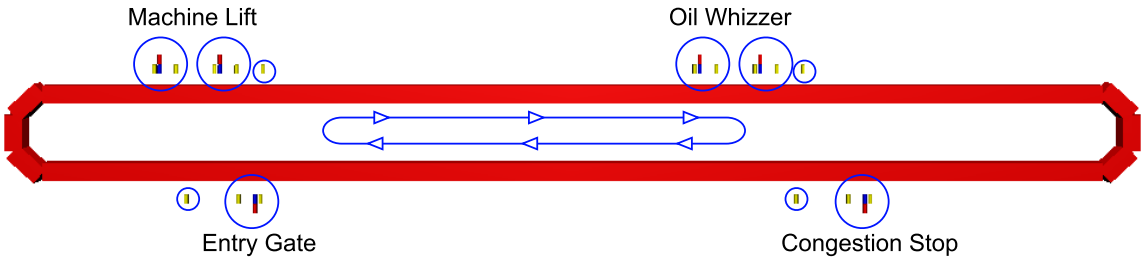


Figure 8.10: Overview of the transportation subsystem. The circled elements are stopper units and are explained in the text. An enlarged view of such a unit can be seen in Figure 8.12.

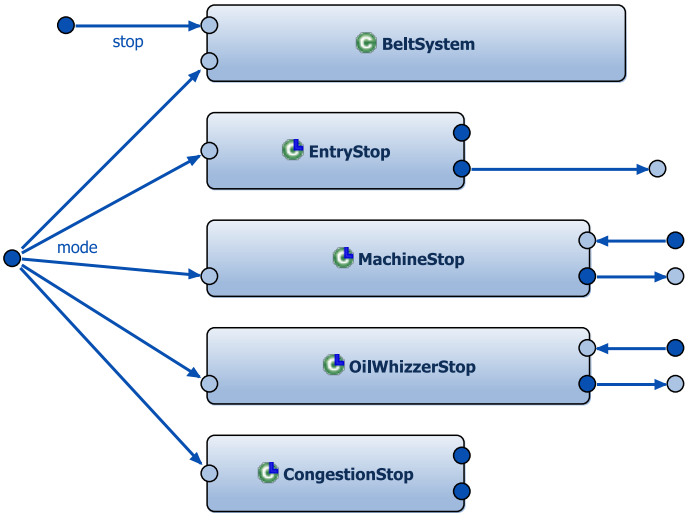


Figure 8.11: The structure of the *Transportation* component.

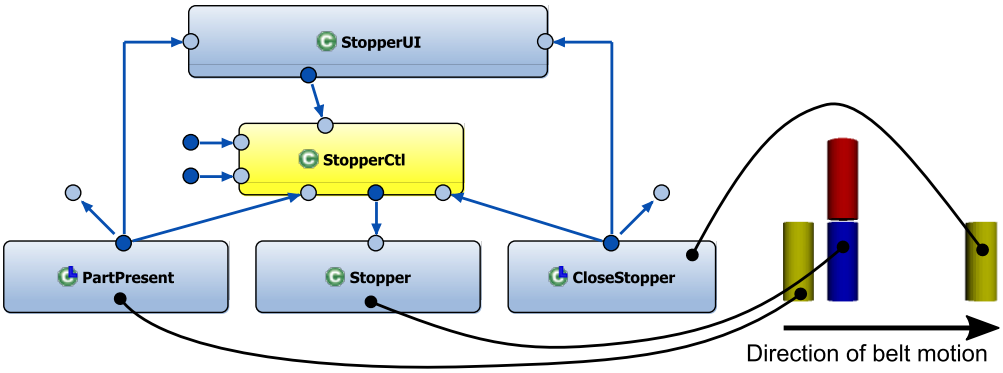
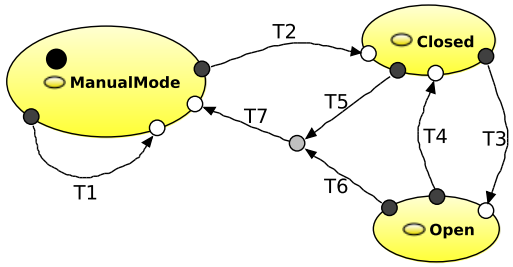


Figure 8.12: Structure of a single stopper unit with corresponding geometry.



T1	**in:**	mode?manual; openUI?_O
	out:	open!_O
T2	**in:**	mode?automatic
	out:	open!**false**
T3	**in:**	mode?automatic; part_present ?**true**; close_stopper ?**false**; request ?**true**
	out:	open!**true**
T4	**in:**	mode?automatic; close_stopper ?**true**
	out:	open!**false**
T5	**pre:**	**true**
T6	**pre:**	**true**
T7	**in:**	mode?manual

Figure 8.13: The automaton used for controlling a single stopper unit.

All of the stoppers are surrounded by two sensors each. As the stopper and its two associated sensors build a logical unit, we modeled them as a reusable mechatronic component in our library. The component (called *StopperUnit*) is shown in Figure 8.12 and consists of three components for the hardware (the sensors called *PartPresent* and *CloseStopper*, and the *Stopper*), one local controller component *StopperCtl* (which is modeled as weakly causal here and thus drawn yellow), and the *StopperUI* which contains the part of the operator panel used to control the stopper in manual mode. Most of the logic is realized in the *StopperCtl*, whose automaton is shown in Figure 8.13. The automaton can be either in manual mode or automatic mode. Mode switching is based on an external signal (transitions T2, T5/T6/T7). In manual mode the position of the stopper (open or closed) is directly controlled by the user via the signal *openUI*, which originates from the *StopperUI* component (transition T1). In automatic mode the position of the stopper is determined by three signals: *part_present* and *close_stopper* are the detection signals from the two sensors, while *request* is an external input signal for the stopper unit. The logic in automatic mode (encoded in transitions T3 and T4) is to keep the stopper closed and only open it, if all three of the following conditions are true:

- a carrier is currently waiting in front of the stopper (part_present ?**true**),
- there is no carrier which could be blocking once the stopper is opened (close_stopper ?**false**), and
- a new carrier was actually requested (request ?**true**).

The stopper is closed again as soon as the *close_stopper* signal becomes true, as this is the earliest moment when the stopper will not block the carrier it just released. The position of the sensors of course has to be adjusted for the geometry of the carrier to work correctly.

After understanding the *StopperUnit*, the remainder of the transportation system is easier to explain. The entry gate position consists of a single *StopperUnit*. The *request* signal for this unit is controlled by both the sensor later on on the belt (indicating possible congestion) and additional checks on the entry gate. The congestion stop works identical to the *StopperUnit* at the entry gate, but only opens based on the following congestion sensor (without additional checks). The reason for this stop is that the transportation belt is never switched off during operation and thus moves below stopped carriers. Thus, the carriers cause a certain pressure on the stopper, which increases with the number of carriers lining up in front of a stopper. The design of the stoppers does not allow more than 10 carriers to be blocked by a single stopper without damaging the part, thus the long area of the transportation belt where more than 10 carriers would fit in has to be split by the additional congestion stop to reduce back pressure. The remaining stoppers are used for the hand-over points to the machine tool and the oil whizzer. Both require two stopper units each; one to keep other carriers waiting until the lift to the machine or whizzer is free, and one to keep the carrier on the lifting position until the lift is activated (and thus the carrier loses contact with the belt). The request signals for these *StopperUnits* are controlled by a combination of signals from the machine/whizzer and the following sensors (indicating free space on the belt).

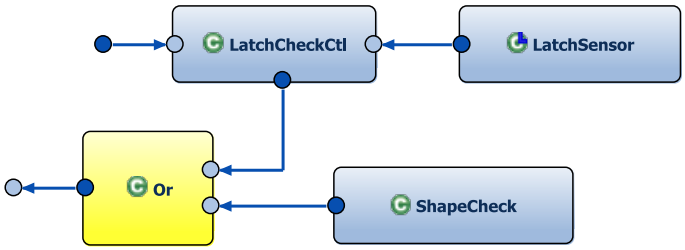


Figure 8.14: Structure of the *EntryCheck* component.

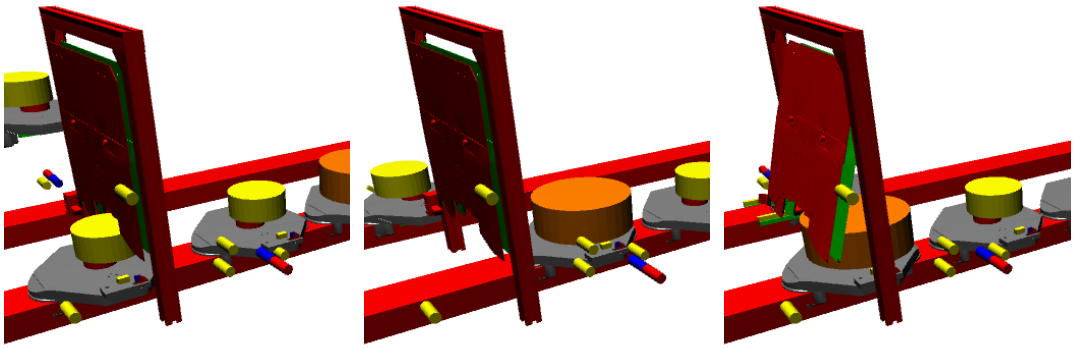
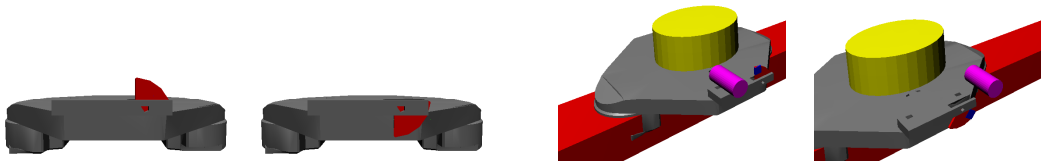


Figure 8.15: Simulation of the size check at the entry gate.

Model Details Two details of the model are interesting, as they involve purely mechanical interaction without interaction of a controller. Both are related to the entry gate, which is realized by the *EntryCheck* component shown in Figure 8.14. The gate performs two checks for every carrier and gear entering the closed part of the loading system. One is to avoid gears of invalid size to enter and the second condition is to avoid gears which already have been processed once to reenter. Both conditions would destroy the gear, and thus the entire transportation belt is stopped (and the operator notified) if one of the checks fails.

To avoid gears of invalid size to enter the machine, a metal plate with a hole of the size of the gear is mounted, such that the *correct* gears can easily pass. Gears which are too large will collide with this plate and cause it to rotate around its hanging attachment at the top. This movement is then detected by a proximity sensor and causes the belt to be stopped. An example for this is shown in Figure 8.15, where the gears are represented by cylinders of yellow (normal size) or orange (too large) color. To recognize the motion of the metal plate, a detector with the same shape and position as the metal plate was used. This sensor causes the plate to rotate as long as something collides with the detector. The solution with the metal plate can only recognize gears which are too large in at least one dimension, but does not prevent smaller gears or empty carriers from entering. According to Kapp engineers, the solution was chosen as smaller gears can be easily detected at the hand-over point to the machine when a gripper tries to pick the gear. While large gears might be damaged by



(a) The gear carrier with the (red) latch in the *up* position and the *down* position.

(b) Simulation run showing the switching of the latch.

Figure 8.16: The latch used for indicating unprocessed gears (*up* position).

the picker, smaller ones are not, but are recognized and can be skipped. Of course, a more complex setup for the entry gate (possibly including measuring equipment) could solve this issue, but at higher cost.

The second problem is to keep processed gears from entering the machine. While the difference between a gear before and after polishing is easily seen visually by the operator, it is hard to detect it using (possibly cheap) sensor equipment. The solution chosen by Kapp is to encode the processing status *mechanically* in the carrier. For this the carrier has a latch, which can be seen in Figure 8.16(a). This latch is moved to its upper position by the operator when mounting a gear to the carrier. Somewhere in the closed region of the loading system, a small metal cylinder is mounted such that the latch is tilted to its down position when it moves by (this is the *LatchCatcher* component mentioned initially). Now the entry gate only has to check whether the latch is in the up position, which can be performed by a simple proximity sensor. If it is down, either the carrier completed a full round on the belt (and thus the gear should have been processed), or the operator forgot to setup the latch. In both cases the belt is stopped to allow the operator to correct the situation. For modeling, the interesting part again is the mechanical interaction between the latch and the metal cylinder used to tilt it down. This is solved in the model by a small detector associated with the latch that detects parts which are directly in front of the latch and cause the latch to rotate to its down position.

Results All relevant parts of the loading system could be modeled using our approach. The approach of modeling mechanical parts and related software within the same (mechatronic) components, such as with the *StopperUnit*, helped to identify and create reusable components. As with the Heller case study, the model was successfully combined with the slightly modified controller software in a virtual commissioning setup. The modifications of the software were required to compensate for the missing parts (machine tool and oil whizzer).

The Kapp example also demonstrates possible shortcomings of our model. As no full physical simulation is involved, the purely mechanical interactions occurring at the entry gate and the latch do not just *happen* during simulation. While the collision detection is part of our semantics, any reaction to a collision that is not just blocking of motion has to be modeled explicitly. However,

this explicit representation in the model also makes this interaction more obvious to an engineer compared to some seemingly random effects which happen in physics simulations.

Based on the model we had some interesting discussion with the control engineers of the (already existing) loading system. Especially the simulation of error scenarios, such as a too large gears which does not pass the entry gate, lead to the exploration of possible strategies for dealing with the problem and also for managing the remaining system parts. One viable strategy is to simply stop the transportation belt, which is the solution we modeled. This however also causes any carriers which already passed the belt to not move on and processing will be paused, although more (valid) gears are ready for processing. Depending on the relative position of the gate and the stopper, the situation could also be resolved by not opening the stopper instead of stopping the belt. For this to work it is crucial that the stopper is behind the entry gate, but not too far so that invalid gears are still accessible to the operator. While in our case the modeled system already existed, this kind of discussion is exactly what we would expect to happen in the design phase during development. The availability of a simulatable model and the option to quickly try different setups both in terms of mechanics and software seems to be valuable for the exchange of ideas and exploration of the solution space.

8.4 Industrial Robots: Wheel Mounting

Differing from the previous two, the third case study is not modeled after an existing system. It was chosen to complement the other examples by featuring industrial robots and being a part of a more traditional assembly line setting. The scenario models the mounting of wheels to a car body and is slightly simplified compared to a real system. The car bodies are transferred along a hanging transportation line, while at both sides the wheels are provided via belt conveyors. Two robots located at both sides of the car body are used to pick the wheels from the belts and fix them on the car bodies.

Model Overview To get a first impression of the model, a screenshot from its simulation is shown in Figure 8.17 and a longer simulation sequence is depicted in Figure 8.18. The model is described by 40 components and 31 automata with 57 atomic states altogether. The top-level structure is defined by the components from Figure 8.19. The main components are the two robots (*FourSegmentRobot*) which are complemented by a separate control component each. The reason for this separation is that the *FourSegmentRobot* models the robots geometry and a generic controller and is a reusable component, while the robot controllers contain the specific motion program for each robot, which differs between both robots. In addition, three components are used to model the conveyors used to transport the car bodies and the supply of wheels for both robots. The components for conveyors and robot controllers are connected to allow a synchronization between transportation and assembly. The unconnected component called *GroundPlane* models the factory floor.

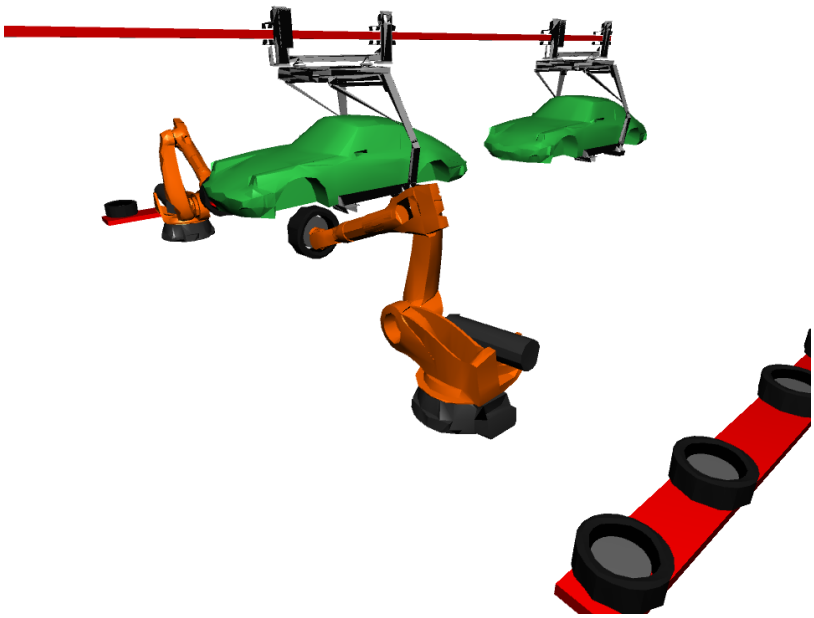


Figure 8.17: Larger view of the wheel mounting model in simulation.

The single robot is described by the *FourSegmentRobot* component, whose decomposition is shown in Figure 8.20. The body of the robot is modeled by the six components *Base*, *Segment1* to *Segment4*, and *Hand*, which each corresponds to one rigid part of the robot and allows for one degree of freedom (excluding the *Hand*). A position of the robot is thus described by a 5-tuple setting values for each of these degrees of freedom. The robot control is realized by two simple components. One is used to split such a (goal) position tuple into its individual values which are then passed on to the corresponding segment components (*PosSplitter*). The second one does the reverse and merges the *reached* signal of the individual sub-components via conjunction (*ReachMerger*). This *reached* signal indicates that a sub-component has reached its goal position. Thus, the merged signal means that the robot has adjusted its position to the target position. The dashed edges indicate the *mover links* between the components (*c.f.*, Section 6.4.2). This means that for example the *Segment2* component will follow every motion of the *Segment1* component. The geometry of the robot's parts is based on a VRML model of a KUKA robot[5] which was split into separate parts and then imported into the modeling tool.

[5] KUKA is a robot manufacturer which was founded in 1898 in Augsburg as an Acetylene factory. As Acetylene is also used for welding, they moved to the construction of welding equipment and in the early 70s also started to construct robots for welding and other applications. A KUKA robot was chosen as they publish geometry data of their robots in various data formats on their web site (`http://www.kuka-robotics.com/`).

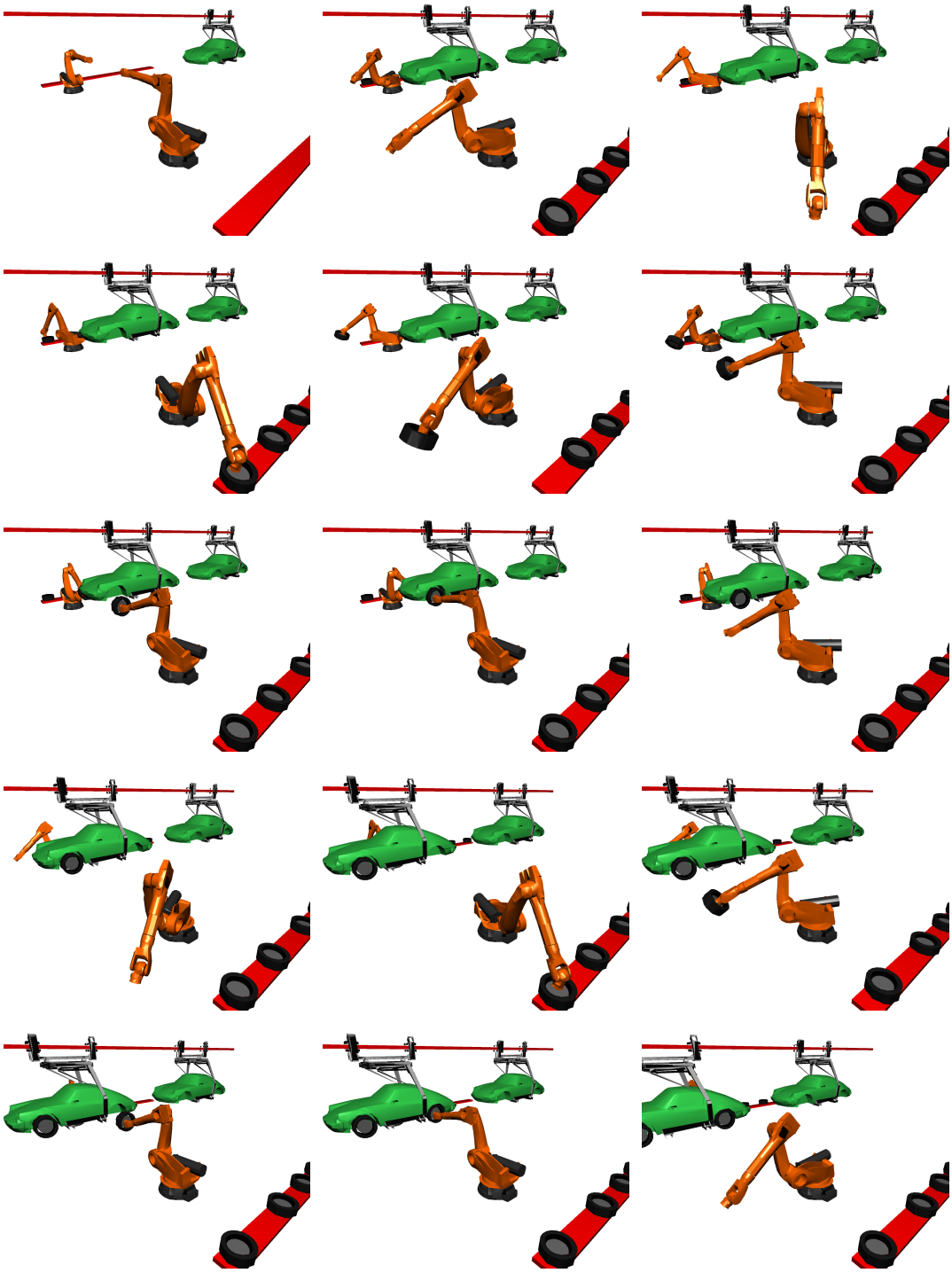


Figure 8.18: Simulation of the wheel mounting example.

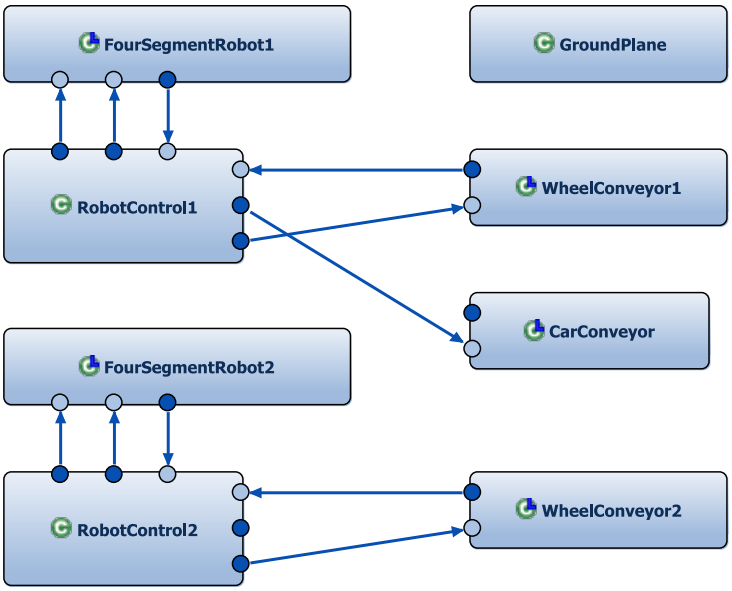


Figure 8.19: Top-level component hierarchy of the wheel mounting example.

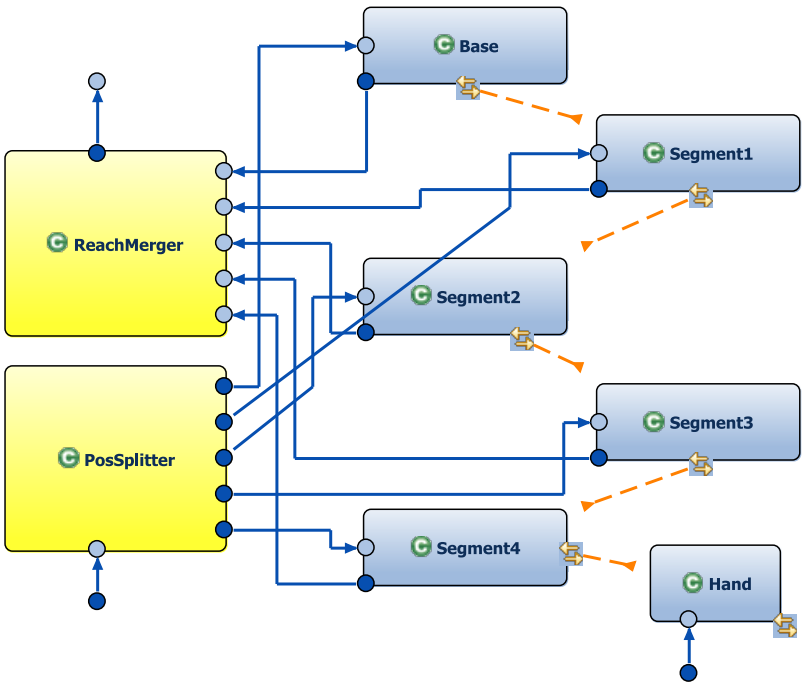
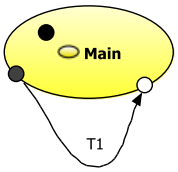


Figure 8.20: Component describing one of the robots.



double _fast = 20;
double _pos = 0;
double _speed = 0;

Main	**deq:**	_mover'=_speed
T1	**in:**	goal?_X
	out:	reached!(_X − _pos < 0.1 && _X − _pos > − 0.1)
	post:	_pos=_pos + _mover; _speed=between((_X − _pos), (− 1.0 ∗ _fast), _fast)

Figure 8.21: Automaton used to control the position of a single robot segment.

Model Details The main challenge for the robots example is robot control. We start our description of the used automata with the ones that control the position of the individual robot segments. Each of these automata affects a single degree of freedom and is nearly identical for all of the segments. The only difference is the speed of movement possible for the joint. Thus, the automaton is a reusable library element with the motion speed being a parameter. One instance of this automaton is shown in Figure 8.21. The automaton consists of the single control state *Main*. Communication with the environment happens via an input port *goal*, over which the goal position is provided (in degree), and the output port *reached*, which indicates that this goal position has been reached. Besides the control state, the automaton has three state variables: _pos stores the current position, _speed stores the current motion speed, and _fast contains the maximal possible absolute motion speed. Actually, _fast is treated as a constant and is never changed. The reason for this constant is that it simplifies exposing the speed as a parameter for technical reasons in our library mechanism. Most of the work happens in transition *T1*. A new target position is read from *goal* (which is always available, as *goal* is a state port). The value for the *reached* port is determined as *true*, if the absolute difference between the current position and the goal position is less than 0.1 degree. This tolerance is used, as it makes the (numeric) simulation of the model more stable in case of rounding errors. But it also resembles the real system, as there always is a certain positioning error depending on the quality of the sensors and drives used. In the postcondition, the variable storing the current position is updated. The variable _mover corresponds to the mover of the part and returns the amount of motion applied in the previous simulation step as explained in Section 6.3.2. The new speed (which is used in the differential equation of the main state to update the mover) is calculated proportionally to the difference between current and target position. The *between* function is a user-defined function specified using the type system, which limits the value of _speed to values between − _fast and _fast . This way the motion will be dampened when the target is near, which avoids moving beyond the target position based on to much momentum. Of course different control strategies would be possible.

The other automaton relevant for controlling the robots is the one for the high-level robot controller components. This automaton provides the target position for all 5 degrees of freedom of a robot. The

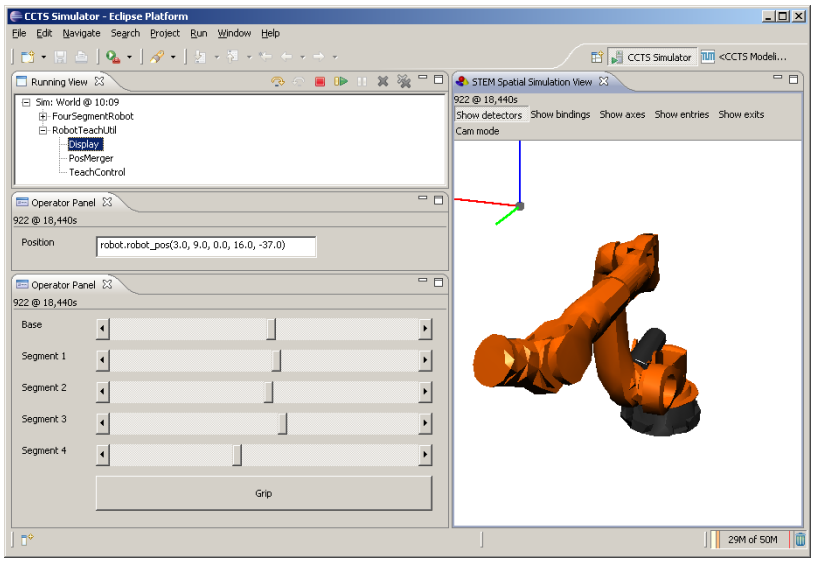
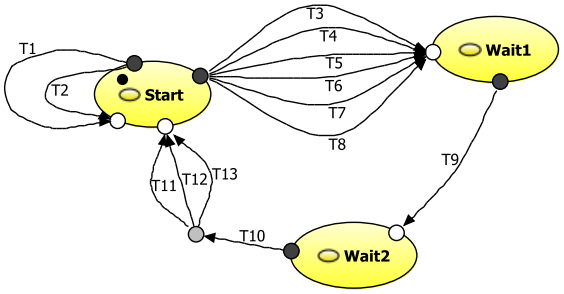


Figure 8.22: Setup used for teaching *robot* positions.

glue between this control automaton which produces 5-tuples, and the individual segment automata which were explained before, are the components *PosSplitter* and *ReachMerger*. These are used to convert the 5-tuple into separate position values and merge the *reached* signals via conjunction. As both are single state automata and fairly simple, they are not shown in detail here. More interesting is the automaton for high-level robot control. Although research is working on smart robot controllers which steer the robot based on camera and sensor input, most robots today are programmed by a process called *teaching*. Teaching means that an operator manually moves the robot to certain positions using a remote control. Each of these positions is stored in terms of joint positions and the final robot program then consecutively moves the robot to each of the recorded positions. Often multiple of such robot programs are stored and can be selected from an external controller.

This type of robot program has been mirrored in our model. For the problem of finding suitable values for all 5 degrees of freedom when performing wheel mounting, the teaching process has been applied using a simulation of the model. For this, a setup similar to the one shown in Figure 8.22 was used, where instead of another controller the robot is directly connected to a *teaching component* which allows direct control of the joints using an operator panel. The goal position sent to the robot can also be seen and recorded for later use in the robot control automaton.

One of the high-level robot control automata is shown in Figure 8.23. The second one is nearly the same but uses different position tuples. The current step in the robot program is managed in the variable _step. The robot program consists of 8 steps which are performed sequentially based on external input. The start of the program is triggered by the *start* signal in transition *T1* which only resets the *done* signal. Transition *T2* realizes the third step, which consists only of closing the gripper. The remaining steps all send a new target position to the robot (transitions *T3* to *T8*).



int _step = 0;

T1	**in:**	start ?**true**
	pre:	_step == 0
	out:	done!**false**
	post:	_step=1
T2	**pre:**	_step == 3
	out:	grip!**true**
	post:	_step=4
T3	**pre:**	_step == 1
	out:	pos!robot.robot_pos (165.0, 26.0, 24.0, 0.0, 40.0)
T4	**pre:**	_step == 2
	out:	pos!robot.robot_pos (165.0, 31.0, 24.0, 0.0, 36.0)
T5	**pre:**	_step == 4
	out:	pos!robot.robot_pos (165.0, 17.0, 24.0, 0.0, 36.0)
T6	**pre:**	_step == 5
	out:	pos!robot.robot_pos (0.0, 0.0, 22.0, 0.0, (− 23.0))
T7	**pre:**	_step == 6
	out:	pos!robot.robot_pos (0.0, 25.0, (− 13.0), 0.0, (− 12.0))
T8	**pre:**	_step == 7
	out:	grip!**false**; pos!robot.robot_pos (0.0, 0.0, 22.0, 0.0, (− 23.0))
T9	**in:**	reached?**false**
T10	**in:**	reached?**true**
	post:	_step=(_step + 1) % 8
T11	**pre:**	_step != 7 && _step != 4
T12	**pre:**	_step == 4
	out:	wheel_taken!**true**
T13	**pre:**	_step == 7
	out:	done!**true**

Figure 8.23: Automaton used to control one of the robots.

Waiting for the robot to reach this position is performed by first waiting for the *reach* signal to be *false* (*T9*) and then *true* again (*T10*). This two step process is required to detect a *rising edge* in the *reach* signal and not just catch an old *true* value. *T10* also increases the current step number (modulo 8). The remaining transitions are used for sending the signals *wheel_taken* and *done* at the appropriate times.

The remaining automata used for the conveyors are fairly simple. They move the belt based on the current signals from the robots and stop the goods (car body or wheel) based on light barriers used to detect their positions. They also emit signals when a wheel or car body is at the correct position, which are then used for starting the robot program.

Results The primary goal of the wheel mounting example was to evaluate how well industrial robots can be included in our mechatronic model. It turned out that once the geometry is prepared the modeling of a single robot is fairly easy. Using the library mechanism, this single robot can then be used multiple times. The main challenge is the high-level controlling of the robot, as it requires position values for all its joints. While the application of *teaching* helped in finding suitable robot positions, the user interface with its sliders is not the best for this task. In a non-prototypical engineering tool, extensions to the user interface should improve this situation. However, both our mechatronic model and tool are well suited for systems involving industrial robots.

8.5 Summary

In this chapter three non-trivial models for mechatronic systems from the automation domain have been presented. Two of them have been proposed from industrial partners and model existing systems, a third one involves robots modeled after those typically found in assembly lines. The examples cover various domain-specific problems, namely tool management, machine loading and unloading, and product assembly. Thus, the case studies are representative for the automation domain, although they can not capture all effects found in the rich field of automation machines.

The case studies demonstrate that all of the systems and the effects relevant for their operation on a logical level can be captured by the operationalized model using our tool. As the modeling theory from Chapter 5 is a superset of the operationalized model, we can also conclude that the modeling theory is sufficiently expressive to describe logical behavior models of these systems. This includes purely mechanical interaction, such as seen by the loading/unloading system, whose underlying logic can be explicitly modeled. The practical applicability of these models during the system development process has been demonstrated for the task of virtual commissioning. The modeling effort required for each of the three examples is in the order of a couple of hours, thus the model is suitable for rapid prototyping of design ideas at an early stage.

9 Summary and Outlook

This concluding chapter summarizes the central contributions and results of this thesis and discusses directions for possible future research. We start with the summary in Section 9.1. To better structure these open topics, they are organized into separate sections, which cover the improvements in the area of engineering support (Section 9.2), formal analysis based on these models (Section 9.3) and further generalization of the model (Section 9.4).

9.1 Summary

None of the existing models allows to formally capture the logical behavior of space-intensive mechatronic systems, as they are common in the automation domain. As shown in Chapter 3, the models either concentrate on one aspect of the system only, behavior *or* space, do not support formal semantics, interfaces, and compositional modeling, or show weaknesses in the space and material models employed. This lack of suitable models at the *logical layer* has several negative consequences in systems engineering, such as (too) late identification of errors in the system's design or the choice of a non-optimal distribution of functionality between mechanics, electrics, and software.

This thesis fills this gap by proposing a novel integrated modeling theory for the specification of mechatronic systems, especially in the context of industrial automation. It is based on a formal model of space and integrates it with the stream-based behavior model provided by FOCUS. The model provides a space-extended notion of components and interfaces, which allows the construction of system models by hierarchical composition of multiple components. In contrast to other models, our composition operator respects and preserves spatial properties, such as relative position and kinematics, as well as the material flow within the system, without the need to adjust the composed elements.

From our modeling theory we derived an operationalization, which allows the application of the theory to non-trivial models. The operationalization provides a concrete meta-model with a graphical and textual syntax that can be used for modeling. The semantics of the modeling theory is directly applicable to the operationalized model. Where the theory allows different possible alternatives in semantic details, our operationalization eliminates this variation, as required for practical application. Besides the concrete syntax, the operationalization also shows how different description techniques, such as automaton and table based approaches, can be combined in a single model, thus allowing the inclusion of different problem specific modeling techniques in the same model.

As the theory defines components only by their syntactic and semantic interfaces, encapsulation is automatically enforced. This ensures, that no component has implicit dependencies to other components, which is crucial for modular development of the models. More concretely, the model allows independent modeling of the components involved, the reuse of individual components in other models, and the construction of libraries of reusable components. This composability extends to the material flow, as components can be described independently of the material they transport or process.

The operationalized model is implemented in a tool prototype, which allows the creation, modification, and simulation of the models. This is the basic functionality required to use our logical behavior models in practice, for example during the exploration of design alternatives, the inspection of specific solutions, or as a means to document and communicate the logical processes and dependencies within a machine. In addition, the tool is used to exemplify structured reuse of parameterizable components, the augmentation of the model with technical detail information, a basic tracing concept to technical (CAD) models, and the use of the model in the automatic generation of simulation models for virtual commissioning. Each of these activities can support the mechatronic development process and thus demonstrate the benefits from using our behavior model during systems development.

To evaluate whether our model is sufficiently expressive to describe realistic systems, we conducted three case studies. The goal of all three was to model a part of a machine tool or factory automation system. Two of the systems are existing systems contributed by industrial partners, while the third one is patterned after an assembly line without an actual industrial realization. The case studies demonstrated, that our model allows to capture all relevant details of the systems while being abstract enough to create the models in only a couple of hours. For the two industrial examples, discussions with the engineers of the corresponding companies demonstrated the usefulness of our abstract models in the understanding of the chain of actions occurring in the machine. Based on the models, different strategies for dealing with certain error conditions were discussed during these meetings. Additionally, virtual commissioning models for the two industrial models were generated. These virtual commissioning models were generated only based on information stored in the behavior model (enriched only by I/O addresses) and were able to run in conjunction with the real controller software of the machines. This again supports our claim that all relevant properties of the systems could be modeled with our approach.

The arguments and findings provided by this thesis support our proposition that our technique is a suitable model for describing the behavior space-intensive mechatronic systems at the logical layer and that the systems engineering process can, in fact, benefit from the utilization of such a model. Of course the ultimate answer to the practical usefulness of our approach can only be given by its application during the development of a system under realistic conditions. This, however, is beyond the scope of this thesis, although it is an interesting direction for future work. There is also a multitude of other open questions for further investigation, which are summarized and explained in the following sections.

9.2 Towards Better Engineering Support

One possible direction of future research is to use the model described in this thesis to support the engineering process beyond modeling/simulation and the approaches described in Chapter 7 (parameterizable libraries, technical augmentation, tracing, virtual commissioning). This includes both the tailoring of the description technique to be better suited to certain domains, as well as the generation of and interaction with other development artifacts.

Alternative and Domain-Specific Description Techniques Our modeling theory is built upon stream processing functions and their composition. As such, the framework supports the use of any description technique for atomic components as long as it can be mapped to these functions. We can also use different descriptions in the same model, *i.e.*, for each component choose the best fitting, as the underlying theory ensures composability. In the operationalization we proposed, automata are the primary description technique, although we present alternatives, such as graphical functions, tabular techniques, and operator panels.

While automata are commonly used in software engineering, electric and mechanical engineers are less used to them. As the model is intended to be used in interdisciplinary mechatronic teams, other description techniques might be better suited. For example, the IEC 61131-3 [IEC03] proposes different languages for PLC programming; one of them, called ladder diagram, resembles electric wire plans and targets electric engineers. Another interesting property of the IEC 61131-3 languages is that as long as only a certain subset is used, subroutines realized in any of these languages can be transformed to any other of the languages without loss of information. This allows each engineer to read and write the program in his preferred language. In parallel to these languages, there are two possible research directions. One is to find a description technique for the behavior of a component that is equally understandable to engineers from all disciplines. This, however, might be difficult as the intersection of modeling knowledge of all those disciplines does not seem very large. So, the alternative would be to define a set of modeling languages, together with transformation rules that can convert models between all of these languages. The set of languages has to be chosen in a way that satisfies all of the involved disciplines. More important are the transformations, which not only should preserve semantics, but also ensure that the transformed models are still understandable. The later point is crucial, as it is easy to *somehow* encode, for example, an automata in a system of (differential) equations, but the new encoding could be meaningless to a human developer.

Another aspect to consider are more domain specific models. Instead of tailoring the languages to the engineers of the different disciplines, we could instead focus on the different application domains. A language suitable for describing sensors and actors would probably look different from one that aims at the programming of industrial robots or a high-level coordinator component. Such work should be based on existing modeling techniques found in these domains. The main task is expected to be the choice of a semantics that both captures the intent of the original language and also fits into the stream-based framework.

Code Generation in the Automation Domain We already described the generation of simulation models for virtual commissioning from our behavior model. These simulation models capture the parts of the system that do not belong to the controller. A natural complement is the generation of the code for these controllers. While code generation from diverse models has been the subject of many research projects, there are some specifics in the automation domain that can affect the generation process.

One difference is the still very common separation of controllers into PLC (Programmable Logic Controllers) and NC (Numerical Control). While the first is specialized in digital input/output operations, logical operations, and timer management, the NC is used for complex continuous control tasks, such as synchronizing movement of several controlled axes. These two main types of controllers are complemented by more specialized devices, such as variable-frequency drives which are used to control the speed of electric motors. While these devices are not freely programmable, their parameterization has a significant effect in the functions of the system. A code generator thus should support the separation of the controller logic to one or more PLCs and NCs, and also account for parameterizable devices. To what degree the separation and configuration process can be automated, and how much manual interaction is required is one of the open research questions.

Another aspect making code generation in the automation domain different is error handling. In many software systems a major part of the code does not provide the primary functionality but rather deals with detection and handling of errors. For embedded systems this fraction is often even larger, as error conditions may lead to safety critical situations and a reboot is usually not a viable alternative. Additionally, compared to critical business systems, developer support is not easily available during operation (*e.g.*, in an airplane). For both error detection and handling there are well-known strategies. Some are common over all domains of embedded and mechatronic systems, such as sensor data plausibilization or using time-outs to check for expected signals and conditions. Other strategies are specific to automation engineering. For example, a usual way to deal with an error is to perform an emergency shutdown, which effectively switches off the power supply of major parts of the system. While this is typically infeasible in automotive and avionic systems, the dynamic forces in automation systems are usually small enough to allow for such a behavior. An important task after an emergency shutdown, which can also be triggered by an operator, is to recover to a known state, as often the current machine state is unknown after a shutdown. A domain specific code generator should be aware of these strategies for error detection, error handling, and error recovery. With this information the programmer could be disburdened from specifying all of the error handling details. Instead, the generator should include them based on the specified *normal behavior* and additional high-level instructions on the general error handling strategy.

Exchange with Mechanical Simulation Models One major application of simulation in mechanical engineering is the calculation of forces and momenta in a mechanical system. This data can then be used to choose sufficiently strong electric drives, or reinforce the overall construction in places exposed to strong forces. The most well-known methods used are multibody systems, where

systems of interconnected rigid bodies are simulated, and the finite element method (FEM) which can be applied to determine the forces and deformations *within* a rigid body.

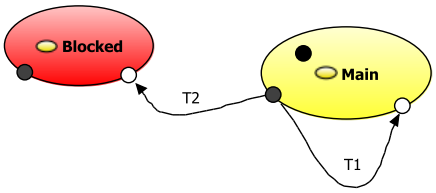
While these approaches work well for most static and purely mechanical systems, in a complex mechatronic system the actual forces applied also depend on additional factors, such as decisions made by the controller and (in the case of automation systems) on the kind and frequency of material inserted into the system. Using the maximal possible forces for the mechanical simulation models can be an overly pessimistic assumption that leads to oversized drives and unnecessarily strong constructions, which both increase the development and assembly cost of the system. Furthermore, mechatronics offers solution alternatives to problems found during simulation. Instead of reinforcing weak spots of the construction, strategies for avoiding these situations can be implemented in the controller. Such strategies could reroute or delay the material stream to smooth peaks in the material insertion frequency, or apply an alternate motion path to a robot if an especially heavy part has been detected. However, without including the controller behavior in the simulation model these solution strategies can not be validated using simulation.

A very tempting solution would be to create a model that incorporates besides our approach also the multibody systems and FEM approaches. However, it seems doubtful that this can be achieved without sacrificing formal semantics or overloading the model with too many details. Additionally, FEM is usually applied only to a limited part of a system to keep it computationally feasible, while our models usually capture a larger scope. A more practical approach would be the exchange of data between all these models to incrementally refine them. From the abstract behavior model we can retrieve the overall setup of the system and both typical and extreme configurations of the mechanical parts as input to mechanical simulations. From these simulations we can receive more detailed motion data and an assessment of the criticality for certain mechanical configurations. This information can then be used to refine the behavior model and possibly add assertions to document unwanted system configurations. This process of simulation and data exchange can then be iterated until a suitable solution is found.

9.3 Towards Analysis of the Models

A natural step in the context of semantically founded behavior models is the application of formal analysis and verification techniques to these models.

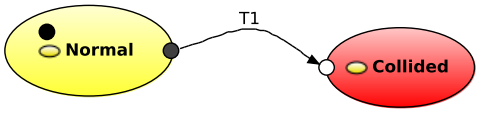
Assertions A prerequisite for an formal analysis is the ability to formulate assumptions or properties which should be verified or analyzed. Properties that are expected to be valid for a model can be captured using assertions. There are two types of assertions: those that have to hold for every



double _fast = 20;
double _pos = 0;
double _speed = 0;

Main	**deq:**	_mover'=_speed
T1	**in:**	goal?_X
	pre:	!(_speed != 0 && _mover == 0)
	out:	reached !(_X − _pos < 0.1 && _X − _pos > − 0.1)
	post:	_pos=_pos + _mover; _speed=between((_X − _pos), (− 1.0 ∗ _fast), _fast)
T2	**pre:**	_speed != 0.0 && _mover == 0.0

Figure 9.1: Automaton used to control the position of a single robot segment.



T1 **pre:** _ground_detector

Figure 9.2: Automaton used to model the assertion for the *GroundPlane*.

model and those that are specific to an individual model. The first class resembles the *world assumptions* discussed in Chapter 5 and typically is enforced by the execution semantics of the model[1]. Here we concentrate on model specific assertions, which have to be contributed by the modeler. Often assertions are expressed using separate languages, such as temporal logics or specification languages based on these (*e.g.*, SALT [BLS06]). Although there are so called spatio-temporal logics [GKK+05, KKWZ07], they are not directly applicable to our behavior models, as we employ a slightly different notion of space.

One possible solution uses direct annotations in the model to mark invalid states. For this, the control states of an automaton can be marked *invalid* and reaching such a state indicates the violation of an assertion. The main benefit of this solution is that the extension of the meta-model is minimal and checking for assertion violations in the simulator is trivial. Still, the mechanism is sufficiently expressive, as the transitions leading to these invalid states may contain complex expressions and be based on additional detectors in the model.

An example is shown in Figure 9.1 which extends the robot segment control automaton from Figure 8.21. The *Blocked* state is only reached when the _speed variable has a non-zero value while the motion applied is 0 (ensured by the pre-conditions of *T1* and *T2*). This condition means that while the segment should have moved it did not. This can only happen due to other parts blocking the

[1] Assertions based on these may be interesting when testing the simulator, which might by accident deviate from the model's semantics.

motion of the robot. As in this model we do not want the robots to collide with any other objects, the *Blocked* state was modeled as invalid (and is thus depicted in red). This shows, how the invalid states can be integrated into existing control automata. Other examples of typical properties that can be modeled this way include areas in which (at certain times) no part may be contained in or certain patterns in the material stream produced by the system. In both cases, the automaton has to use one or more detectors to also include spatial properties in the assertion. An example for this case is the assertion that no wheel may ever hit the ground in the wheel mounting example from Section 8.4. This is expressed by a large detector representing the floor. This detector belongs to the *GroundPlane* component from Figure 8.19 which has the automaton from Figure 9.2 assigned. The automaton is very simple and just switches to the (invalid) *Collided* state as soon as the detector (called _ground_detector) is activated, *i.e.*, some part collides with it. This assertion demonstrates the benefits of including geometry into the model, as it is fairly easy to describe using a detector.

Theorem Proving and Model Checking The ultimate goal in formal analysis and verification is to give a proof for the correctness of a *system*. Typically, however, the proofs are only applied to a model of the system. In our context we would for example check that for all inputs none of the assertions are violated. The most general and at the same time most expensive technique is theorem proving, usually supported by an interactive theorem prover. Here the proof is constructed mostly manually and checked for correctness by the theorem prover. In addition, the tool can find partial proofs automatically, thus unburdening the user from some of the work. Examples for this technique using the interactive theorem prover *Isabelle* [NPW02] on embedded systems are reported in [BGH$^+$06, BBG$^+$08]. While this technique requires too much effort to be used for entire systems, its application to a very narrow but critical subsystem can be worthwhile. Thus, one possible direction of future research is to develop techniques for the transformation of our models into formulas that can be understood by a theorem prover, and the mechanization of the mathematical theory required to perform property proofs for these models.

An alternative proof technique for state-based systems is model checking [CGP99]. It promises fully automated proofs without manual interaction and works by an exhaustive search of the state space. As the state space can be huge, its naive exploration is usually not feasible. To reduce the state space, techniques such as partial order reduction and bitstate hashing can be applied [Hol04]. Another approach is to encode the state space and the possible transitions as binary decision diagrams and evaluate the properties on this *symbolic* representation [McM93], or only evaluate the property for finite runs of the system (bounded model checking [BCC$^+$03]).

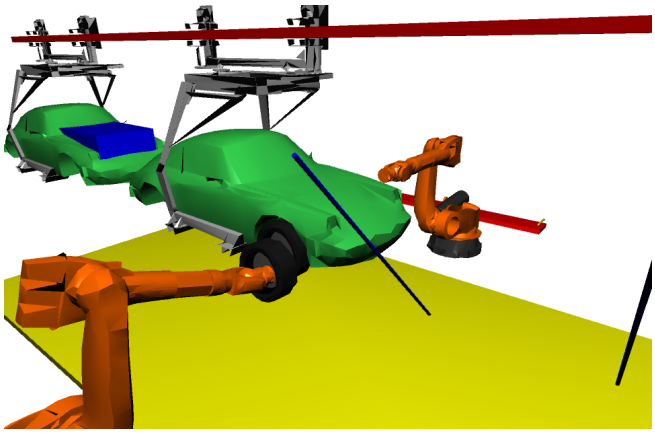
In the case of timed and hybrid systems, the state space is infinite due to the use of continuous variables. Techniques that apply model checking to models of such systems work by dividing the state space to a finite number of subsets that capture states that exhibit equivalent future traces [ACD93, HHWT97]. In the case of hybrid systems, this technique is only applicable to certain classes of hybrid systems, as most properties are undecidable for general hybrid systems [ACH$^+$95].

The obvious open question is whether we can apply model checking to our models of space-intensive mechatronic systems. The answer to this question also depends on the model of space (transformable collision space, *c.f.*, Section 4.2.2) and time we are using. It turns out that for the space and time model we chose for our operationalization, there is no embedding in either discrete state-based systems (due to the continuous positions of objects), nor in any of the subclasses of hybrid systems that are known to be model checkable. One obstacle for the embedding in restricted classes of hybrid systems is our usage of Euclidean space with arbitrary rotations. The formulas involved in expressing transformations and collisions are outside of the typically studied classes of expressions. So there are two open questions. The first is, whether there is a restricted class of hybrid systems which allows for an embedding of our operationalized model while still being model checkable. The dual question is, whether there is a restricted transformable collision space that allows an embedding into any model checkable system model, while still being sufficiently expressive to capture the central aspects of realistic systems.

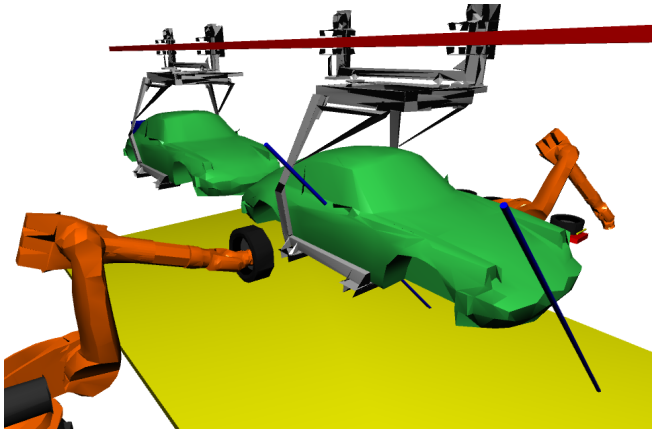
Random Testing To contrast the *heavy weight* approaches of theorem proving and model checking, we next discuss random testing of the models, for which we already have first results. Testing is the process of executing a system with certain inputs to find deviations from the intended behavior. As such, "testing can be a very effective way to show the presence of bugs, but it is hopelessly inadequate for showing their absence" [Dij72]. Still, it can be useful in practice to apply testing techniques as they are less complex and resource intensive compared to more formal approaches.

Different from the usual model-based testing process, we do not target the testing of the actual system, but rather of the system's design model. The goal is to find logical errors in the behavior model at an early stage of the development process, as solutions which require changes to the system, such as additional sensors, are easier to perform then. Thus, we want to find inputs which are used to simulate the behavior model and check whether certain assertions are violated during this simulation. The technique applied is *random testing* which generates suitable inputs randomly based on certain probability distributions. The benefit of random testing is that it is comparatively easy to realize and is not affected by the state explosion problem, as many other techniques. Despite its simplicity, for programs random testing can find a large class of problems [PLEB07, CMOP08].

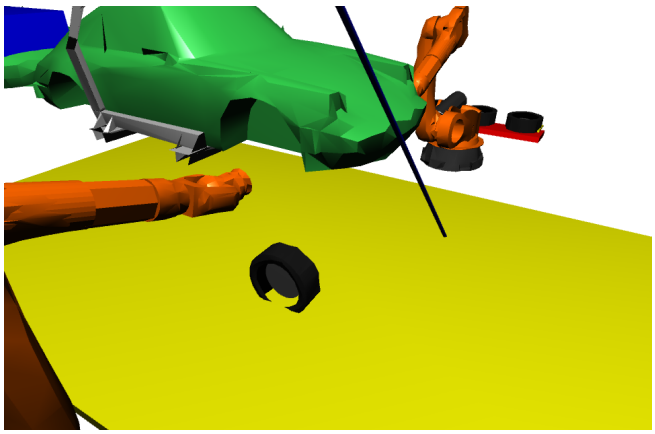
As a proof-of-concept, we implemented support for random testing in the tool and inserted a bug into the wheel mounting model from Section 8.4. The bug is already visible in Figure 8.19 as only *RobotControl1* is connected to the *CarConveyor* but not *RobotControl2*. Thus the conveyor used for the cars is only synchronized to the first robot but not to the second one. During normal operation this problem does not even have an effect as both robots perform the same task and will usually mirror each others movement. An error only occurs if one of the conveyors delivering wheels to the robots runs out of wheels, as then the robot will wait for new wheels to appear and the robots no longer run in synchrony. The application of random testing to the model resulted in several execution traces that demonstrate errors in the model. Interestingly, not all of these errors were due to the manually inserted bug, but also had causes in model parts which were assumed to be free of errors.



(a) Trying to mount two wheels to the same car axis.



(b) Missing to mount a wheel to the axis.



(c) Dropping a wheel to the ground.

Figure 9.3: Errors in the robot example found via random testing. The long cylinders are light barriers used to track the car positions.

Examples of the errors found are shown in Figure 9.3. The first error (Figure 9.3(a)) causes two wheels to be placed to the same axis of the car. The reason is that the car is only advanced forward if the first robot has mounted its wheel. If the first robot runs out of wheels the second robot will continue mounting wheels to the same position, which causes the robot to block and violate the non-blocking assertion. The converse situation leads to the error from Figure 9.3(b). There the second robot runs out of wheels, but as the first robot already mounted its wheel the car is advanced on the belt. When later wheels are available for the second robot it tries to mount the wheel to an invalid position, causing a collision between the wheel and the car's body. A violation of the *GroundPlane's* assertion is shown in Figure 9.3(c). There the robot started to move the wheel to its target position although no car was available yet. After releasing the wheel, it moves towards the floor and causes a collision, *i.e.*, violates the assertion.

Even in our simple example random testing found various (and partially unexpected) bugs in the model. While random testing will usually not find problems which only occur after a sequence of very specific inputs, many problems can be found using this approach. As the testing process can happen without supervision and recognize errors due to the presence of assertions in the model, it can be applied very cost effective, for example in a nightly run. Of course the results from our simple example can not be generalized, so one important future step is the evaluation of the approach for more complex and realistic models. Another open topic is the refinement of the approach using ideas from random testing of programs, such as including the results from earlier test runs to direct the choice of test inputs [PLEB07]. Finally, it would also be interesting to apply the test cases that were found on the model level to the real system. The difference in abstraction between both is the same as for virtual commissioning, so an approach similar to that from Section 7.4 can probably be applied.

9.4 Towards General Mechatronic Systems

The model we presented in this thesis has a specific focus on the inclusion of spatial properties. While these properties turn out to be important for mechatronic systems in the automation domain, there are of course systems (both within and outside of this domain) whose functionality is realized by other physical effects we do not respect in our model. An overview of these limitations was already given in Section 5.5.3. Here we will discuss, how they could be integrated into an abstract formal model with the specific focus of preserving composability. We discuss these effects separately in the next paragraphs without claiming to be exhaustive.

The problem with physical laws in general is not the creation of a model that allows to capture the relevant quantities and a simulation that follows these laws. During the last decades simulation models for nearly all areas of physics and physical effects have been developed. Instead, for a logical model used during development of mechatronic systems, we consider two aspects to be important. The first is the description of the transition between the physical and the logical (or digital) world. As physical simulations often consider purely physical and closed systems, this

interplay with discrete systems is typically not included. Our second claim is that models that are used for solution design should enforce encapsulation of components and make interactions explicit. If a certain physical effect is essential for the provisioning of a function or has to be compensated for, then this is exactly the information that should be contained in the model itself and not be an implicit result of seemingly unplanned interactions of components. Thus, the next paragraphs should be read with this intention in mind.

Classical Mechanics Classical mechanics is based on the *position* and *mass* of objects (and implicitly *time*). From these it derives *velocity*, the rate of positional change (first derivative of position), *acceleration*, the rate of change in velocity (second derivative of position), *momentum*, the product of mass and velocity, and *force*, the product of mass and acceleration. For these derived quantities, certain relations hold, such as preservation of momentum.

Our model includes both position and time, but does not capture the mass of objects or their average density (from which mass could be derived as we know their volume). As stated before, we consider encapsulation and explicitness of interaction central for logical behavior models. Thus, fully supporting classical mechanics is not as easy as just recording the mass for each component and implementing the laws as part of the semantics. For example, to make force more explicit, one could annotate components with explicit interaction points, where force may act upon the component or where the component may exercise force on other components. This makes force exchange a part of the component's interface and allows to specify the reactions to force as part of the component's behavior. The semantics could enforce, that a valid execution of the model only involves forces between these interaction points, while forces outside of the specified range are considered violations and hint at cases missed by the modeler. One open design decision of a modeling technique is also, whether force exchange is described purely on a logical level, or if the spatial model presented in this thesis is applied to correlate the interaction points with actual spatial interference.

Thermodynamics The area of thermodynamics studies the conversion between heat (thermic energy) and mechanic work. Typical phenomena studied are the exchange of heat between different objects, and the change of volume or pressure based on temperature changes. Thermodynamic effects can, for example, be relevant in automation systems where deviations have to be kept in the micrometer range, such as placement systems for circuit board assembly with SMDs[2], or high precision grinding machines. For these systems the shape change caused by high temperature can cause errors in the produced goods. Often the system itself is also the source of the heat, as electric drives or the friction during grinding produce thermic energy. In these cases strategies for cooling and to measure and compensate for the shape change should be part of a solution to the engineering problem, and hence a part of a design model.

[2]Surface mount devices (SMDs) are electronic components that are mounted directly on the surface of printed circuit boards. The smallest of these devices measure 0.4 times 0.2 millimeters.

One way to include thermodynamics into the model could be to make the temperature of a component a part of its state. If the behavior function can depend on this temperature, the change in shape can be easily expressed in our theoretical model (the operationalization would need to be extended to allow for shape-changing parts). The exchange of thermic energy between components (which is important to describe cooling) could be formalized by special ports, where a connection of components via these ports indicates that they exchange heat. The spatial model can be used to some extent to check these connections, as components that exchange heat are usually expected to be placed next to each other. The opposite does not hold in general, as some kind of insulation might be in place. The main challenge of integrating this into the semantics is that the direction and amount of energy exchange depends on the state of the components involved, *i.e.*, energy always is transferred from the hotter to the cooler component.

Electromagnetic Effects Electromagnetism describes the interaction and force between electrically charged objects. It can be used to describe and explain both electric and magnetic fields and their relationships, but also electromagnetic radiation, including both radio transmission and visible light. In mechatronic systems, we are not interested in the interactions of individual electrons, but rather on electromagnetic effects on a larger scale. The rotation of a electric drive, the light ray of a photoelectric barrier, or the switching of a relay all depend on electromagnetism. However, it is seldom required to capture these effects on the level of electromagnetic interaction, but rather on a logical level. In fact, all three examples are supported by our model in terms of converting a logical signal to motion, collisions to logical signals, or between logical signals.

There is still one case, where direct inclusion of electromagnetism into the modeling theory can be useful: as a disturbing factor. For example, in the assembly lines of car manufacturers there are many sources for strong electromagnetic effects, such as the drives of heavy robots or high-voltage wires to power welding devices. The full metal car bodies moved through the assembly line further affect these magnetic and electric fields. The inner working of several electric and electronic parts can be changed by these fields, which leads to unexpected overall behavior of the system. As errors are often only induced by very specific configurations and runs of the system, these errors are extremely hard to find in practice. To support the detection and removal of such problems, electromagnetism could be included into the behavior model. The core task for future work in this direction is to find a model of electromagnetism that is sufficiently expressive to describe these effects, but still simple enough to not completely occlude the internal logics of the behavior model. Based on this, supporting processes and techniques for detecting and avoiding errors due to electromagnetism could be developed.

Liquids and Loose Goods While our model is based on rigid discrete material, there are many systems that deal with liquids or with loose goods that behave similar to liquids, such as sand or rice. There are two very common roles of liquids in mechatronic systems. One role is as the medium for the transmission of energy in a hydraulic system, the other is as the processed material, for example in a chemical plant. In the former case, typically the focus is on the amount of energy transferred in

terms of pressure, in the later case usually the actual quantity of liquid is more important. Different from discrete material, that is usually transported freely in the machine, liquids typically follow a more restricted path through pipes and tubes.

The data-flow oriented description approach of our model can to some extent also describe the flow of liquids through the system. However, if liquids play a central role in a plant, it might be better to model the exchange of liquid between components explicitly. This includes both the syntactical view by introducing ports used for the exchange of liquids, and the way of describing the flow not as discrete messages but rather in terms of a flow rate. The channels connecting these ports could then also be bounded, *i.e.*, only allow a certain maximal rate of flow between them. If the model describes both the possible flow rates (depending on the current state of the system) and the controller logic, this allows to analyze the maximal throughput of the system depending on the controller program used.

Abrasion Milling and grinding machines change the shape of material objects (usually made from some kind of metal) by removing their unwanted parts. This process is known as abrasion. In our model we assume that a change in shape is initiated from *within* a component (although it may be triggered by some external event). Contrary, in the abrasion scenario a component (*e.g.*, the grinder) changes the shape of *another* component. One way to describe abrasion in a model would be to allow components to send *negative space* to each other (maybe via special ports). The negative space indicates for example the region covered by the milling tool. The interpretation then is that the volume of a part is defined by the volume defined by the component's behavior *minus* all of the negative spaces received over time. Such a solution would also require an extended model of space, as our transformable collision space does not support spatial subtraction or intersection but only their approximation using a definition based on (non-)collision after subtraction.

During abrasive grinding usually also significant thermal energy is produced, as well as dust and chipping, which all have to be transferred out of the machine. So both the paragraphs on thermodynamics and on liquids and loose goods apply as well.

Summary In this concluding section we outlined various possible extensions to the model that would allow the application of our approach to more general mechatronic systems. This list of course is by far not complete. For example aerodynamics was not discussed, although it is very important in the avionics domain. The main reason is that more experience with these fields of physics and their practical application is required to discuss their inclusion in a model beyond mere speculation. On the other hand the discussion shows that there is ample space for future improvements of the model or similar approaches. However, from a practical perspective care must be taken not to overload the language with too many options and possibilities, as then the models might become too complicated and detailed to be useful. A more promising approach could be a family of modeling languages that can be tailored to the actual problem domain to only include support for these physical effects that are relevant in the domain's context.

9.5 Conclusion

In this chapter we summarized the main results of this thesis and outlined possible directions for future research motivated and enabled by our work. While we solved some open questions with our model, of course many interesting and relevant problems remain. With the steady increase in complexity of the systems built, we expect the role of mechatronics to become even more important than it is today. No engineering discipline can solve the problems involved in developing these machines in isolation. The only remedy is a tight collaboration between these disciplines. Modeling can support such collaboration, by providing a *lingua franca* that allows communication and exchange of ideas between engineers of different disciplines. In addition, modeling provides an abstract view to a system that helps to manage its complexity during design and development. Our work is intended to be one building block on the way to a comprehensive modeling theory of mechatronic systems, which enables the steady progress in the development of the more advanced systems of the future.

Bibliography

[ABH+99] József Albert, Klaus Bender, Thomas Holzmüller, Bernhard Jünger, Oliver Kaiser, Werner Kriesel, Oliver Prinz, Christoph Schaich, Joachim Schullerer, and Jacek Tomaszunas. *Echtzeitsimulation zum Test von Maschinensteuerungen*. Herbert Utz Verlag, 1999.

[ACD93] Rajeev Alur, Costas Courcoubetis, and David L. Dill. Model-checking in dense real-time. *Information and Computation*, 104(1):2–34, 1993.

[ACH+95] Rajeev Alur, Costas Courcoubetis, Nicolas Halbwachs, Thomas A. Henzinger, Pei-Hsin Ho, Xavier Nicollin, Alfredo Olivero, Joseph Sifakis, and Sergio Yovine. The algorithmic analysis of hybrid systems. *Theoretical Computer Science*, 138(1):3–34, 1995.

[AD94] Rajeev Alur and David L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.

[Alt05] Simon L. Altmann. *Rotations, Quaternions, and Double Groups*. Dover Publications, 2005.

[APHvB07] Marco Aiello, Ian Pratt-Hartmann, and Johan van Benthem, editors. *Handbook of Spatial Logics*. Springer, 2007.

[BBG+08] Jewgenij Botaschanjan, Manfred Broy, Alexander Gruler, Alexander Harhurin, Steffen Knapp, Leonid Kof, Wolfgang J. Paul, and Maria Spichkova. On the correctness of upper layers of automotive systems. *Formal Aspects of Computing*, 20(6):637–662, 2008.

[BCC+03] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, Ofer Strichman, and Yunshan Zhu. Bounded model checking. *Advances in Computers*, 58:118–149, 2003.

[BCE+03] Albert Benveniste, Paul Caspi, Stephen A. Edwards, Nicolas Halbwachs, Paul Le Guernic, and Robert de Simone. The synchronous languages 12 years later. *Proceedings of the IEEE*, 91(1):64–83, 2003.

[BD04] Bernd Bruegge and Allen H. Dutoit. *Object-Oriented Software Engineering*. Pearson-/Prentice Hall, second edition, 2004.

[Ben94] Brandon Bennett. Spatial reasoning with propositional logics. In *Proceedings of the 4th International Conference on Knowledge Representation and Reasoning*, 1994.

[Ber07] Gérard Berry. Synchronous design and verification of critical embedded systems using scade and esterel. In *Proceedings of the 12th International Workshop of Formal Methods for Industrial Critical Systems (FMICS'07)*, 2007.

[BFG07] Tibor Bakota, Rudolf Ferenc, and Tibor Gyimothy. Clone smells in software evolution. In *Proceedings of the 23rd IEEE International Conference on Software Maintenance (ICSM'07)*, 2007.

[BFG+08] Manfred Broy, Martin Feilkas, Johannes Grünbauer, Alexander Gruler, Alexander Harhurin, Judith Hartmann, Birgit Penzenstadler, Bernhard Schätz, and Doris Wild. Umfassendes Architekturmodell für das Engineering eingebetteter Software-intensiver Systeme. Technical Report TUM-I0816, Technische Universität München, 2008.

[BFS05] Marcello Bonfé, Cesare Fantuzzi, and Christian Secchi. Unified modeling and verification of logic controllers for physical systems. In *Proceedings of the Joint 44th Conference on Decision and Control, and 2005 European Control Conference (CDC-ECC'05)*, 2005.

[BG05] Sven Burmester and Holger Giese. Visual integration of UML 2.0 and block diagrams for flexible reconfiguration in MECHATRONIC UML. In *Proceedings of IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC'05)*, 2005.

[BGH+06] Jewgenij Botaschanjan, Alexander Gruler, Alexander Harhurin, Leonid Kof, Maria Spichkova, and David Trachtenherz. Towards modularized verification of distributed time-triggered systems. In *Proceedings of the 14th International Symposium on Formal Methods (FM'06)*, 2006.

[BGH+07] Sven Burmester, Holger Giese, Stefan Henkler, Martin Hirsch, Matthias Tichy, Alfonso Gambuzza, Eckehard Müch, and Henner Vöcking. Tool support for developing advanced mechatronic systems: Integrating the fujaba real-time tool suite with camelview. In *Proceedings of the 29th International Conference on Software Engineering (ICSE'07)*, pages 801–804, 2007.

[BH09] Jewgenij Botaschanjan and Benjamin Hummel. Specifying the worst case - orthogonal modelling of hardware errors. In *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis (ISSTA'09)*, 2009.

[BH10] Jewgenij Botaschanjan and Benjamin Hummel. Material flow abstraction of manufacturing systems. In *Proceedings of the 7th International Colloquium on Theoretical Aspects of Computing (ICTAC'10)*, 2010.

[BHH+09] Jewgenij Botaschanjan, Thomas Hensel, Benjamin Hummel, Alexander Lindworsky, and Michael F. Zäh. Simulationsmodelle für die virtuelle Inbetriebnahme. *ATZproduktion*, 05-06:18–22, 2009.

[BHH+10] Jewgenij Botaschanjan, Thomas Hensel, Benjamin Hummel, Alexander Lindworsky, Michael Zäh, Gunther Reinhart, and Manfred Broy. AutoVIBN – Abschlussbericht. Technical Report TUM-I1012, Technische Universität München, 2010. Automatische Generierung von Verhaltensmodellen aus CAD-Daten für die qualitätsorientierte virtuelle Inbetriebnahme.

[BHL09] Jewgenij Botaschanjan, Benjamin Hummel, and Alexander Lindworsky. Interdisziplinäre Funktionsmodellierung im Anlagenbau. *ZWF Zeitschrift für wirtschaftlichen Fabrikbetrieb*, 01-02:71–75, 2009.

[BHLH09] Jewgenij Botaschanjan, Benjamin Hummel, Alexander Lindworsky, and Thomas Hensel. Integrated behavior models for factory automation systems. In *Proceedings of the 14th IEEE International Conference on Emerging Technology and Factory Automation (ETFA'09)*, 2009.

[BHS99] Manfred Broy, Franz Huber, and Bernhard Schätz. AutoFocus – Ein Werkzeugprototyp zur Entwicklung eingebetteter Systeme. *Informatik Forschung und Entwicklung*, 13(13):121–134, 1999.

[BK87] Jan A. Bergstra and Jan Willem Klop. Acp_τ: A universal axiom system for process specification. In *Proceedings of Algebraic Methods: Theory, Tools and Applications*, 1987.

[BKM07] Manfred Broy, Ingolf Krüger, and Michael Meisinger. A formal model of services. *ACM Transactions on Software Engineering Methodology*, 16(1), 2007.

[BLS06] Andreas Bauer, Martin Leucker, and Jonathan Streit. SALT—structured assertion language for temporal logic. In *Proceedings of the 8th International Conference on Formal Engineering Methods (ICFEM'06)*, 2006.

[BM05] Jan A. Bergstra and C. A. Middelburg. Process algebra for hybrid systems. *Theoretical Computer Science*, 335(2-3):215–280, 2005.

[Bro97] Manfred Broy. The specification of system components by state transition diagrams. Technical Report TUM-I9729, Technische Universität München, 1997.

[Bro01] Manfred Broy. Refinement of time. *Theoretical Computer Science*, 253(1):3–26, 2001.

[Bro08] Manfred Broy. *Relating Time and Causality in Interactive Distributed Systems*. IOS Press, 2008. Proceedings of the NATO Advanced Study Institute on Engineering Methods for Software Safety and Security.

[BS01] Manfred Broy and Ketil Stølen. *Specification and Development of Interactive Systems: Focus on Streams, Interfaces, and Refinement*. Springer, 2001.

[CGP99] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. The MIT Press, 1999.

[CMOP08] Ilinca Ciupa, Bertrand Meyer, Manuel Oriol, and Alexander Pretschner. Finding faults: Manual testing vs. random+ testing vs. user reports. In *Proceedings of the 19th International Symposium on Software Reliability Engineering (ISSRE'08)*, 2008.

[CR05] Pieter J. L. Cuijpers and Michel A. Reniers. Hybrid process algebra. *Journal of Logic and Algebraic Programming*, 62(2):191–245, 2005.

[DHJ+08] Florian Deissenboeck, Benjamin Hummel, Elmar Juergens, Bernhard Schätz, Stefan Wagner, Jean-François Girard, and Stefan Teuchert. Clone detection in automotive model-based development. In *Proceedings of the 30th International Conference on Software Engineering (ICSE'08)*, 2008.

[DHJ+10] Florian Deissenboeck, Benjamin Hummel, Elmar Juergens, Michael Pfaehler, and Bernhard Schaetz. Model clone detection in practice. In *Proceedings of the 4th International Workshop on Software Clones (IWSC'10)*, 2010.

[Dij72] Edsger W. Dijkstra. The humble programmer. *Communications of the ACM*, 15(10):859–866, 1972.

[Döb09] Christoph Döbber. Modellierung und Simulation von Bedientafeln zur Steuerung reaktiver Systeme. Bachelor's thesis, Technische Universität München, 2009.

[Dra10] Rainer Draht, editor. *Datenaustausch in der Anlagenplanung mit AutomationML*. Springer, 2010.

[Dro95] R. Geoff Dromey. A model for software product quality. *IEEE Trans. Software Eng.*, 21(2):146–162, 1995.

[Dür99] Rainer Dürr. *Kopplungsansätze mechatronischer Systeme in Modellierung und Simulation*. PhD thesis, Universität Stuttgart, 1999.

[DWP+07] Florian Deissenboeck, Stefan Wagner, Markus Pizka, Stefan Teuchert, and Jean-François Girard. An activity-based quality model for maintainability. In *Proceedings of the 23rd IEEE International Conference on Software Maintenance (ICSM'07)*, 2007.

[EG97] Baher A. El-Geresy. The space algebra: Spatial reasoning without composition tables. In *Proceedings of the 9th International Conference on Tools with Artificial Intelligence (ICTAI'97)*, 1997.

[Eng00] Vadim Engelson. *Tools for Design, Interactive Simulation, and Visualization of Object-Oriented Models in Scientific Computing*. PhD thesis, Linköpings Universitet, 2000.

[ESMS06] Thilo Espenberger, Andreas Sziwek, Heiko Mannheim, and Andreas Schietinger. Detailspezifikation der SmartAutomation-Modellanlage, 2006.

[Fer94] Luca Ferrarini. A theoretical framework to model and analyze manufacturing systems. In *Proceedings of the 33rd Conference on Decision and Control (CDC'94)*, 1994.

[GBSO04] Holger Giese, Sven Burmester, Wilhelm Schäfer, and Oliver Oberschelp. Modular design and verification of component-based mechatronic systems with online-reconfiguration. In *Proceedings of the 12th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE'04)*, 2004.

[Geh05] Matthias Gehrke. *Entwurf mechatronischer Systeme auf Basis von Funktionshierarchien und Systemstrukturen*. PhD thesis, Universität Paderborn, 2005.

[GKK+05] David Gabelaia, Roman Kontchakov, Ágnes Kurucz, Frank Wolter, and Michael Zakharyaschev. Combining spatial and temporal logics: Expressiveness vs. complexity. *Journal of Artificial Intelligence Research*, 23:167–243, 2005.

[Gli07] Martin Glinz. On non-functional requirements. In *Proceedings of the 15th IEEE International Requirements Engineering Conference (RE'07)*, 2007.

[Har87] David Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, 1987.

[HB08] Benjamin Hummel and Peter Braun. Towards an integrated system model for testing and verification of automation machines. In *Proceedings of the 2008 International Workshop on Models in Software Engineering (MiSE'08)*, 2008.

[HCRP91] Nicolas Halbwachs, Paul Caspi, Pascal Raymond, and Daniel Pilaud. The synchronous data flow programming language LUSTRE. *Proceedings of the IEEE*, 79(9):1305–1320, 1991.

[Hen00] Thomas A. Henzinger. The theory of hybrid automata. In *Verification of Digital and Hybrid Systems*, NATO ASI Series F: Computer and Systems Sciences 170, pages 265–292. Springer, 2000.

[HH09] Markus Herrmannsdoerfer and Benjamin Hummel. Library concepts for model reuse. In *Proceedings of the Workshop on Language Descriptions Tools and Applications (LDTA'09)*, 2009.

[HHWT97] Thomas A. Henzinger, Pei-Hsin Ho, and Howard Wong-Toi. HYTECH: A model checker for hybrid systems. In *Proceedings of the 9th International Conference on Computer Aided Verification (CAV'97)*, 1997.

[HM09] Peter Höfner and Bernhard Möller. An algebra of hybrid systems. *Journal of Logic and Algebraic Programming*, 78(2):74–97, 2009.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[Hof93] Christoph M. Hoffmann. *Geometric and solid modeling*. Morgan Kaufmann, 1993.

[Hol04] Gerard Holzmann. *The Spin Model Checker*. Addison-Wesley, 2004.

[Hum09] Benjamin Hummel. A semantic model for computer-based spatio-temporal systems. In *Proceedings of the 16th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems (ECBS'09)*, 2009.

[IEC03] IEC. *IEC 61131-3: Programmable controllers – Part 3: Programming languages*. International Electrotechnical Commission, 2003.

[IEC07] IEC. *IEC 61784-1: Industrial communication networks – Profiles – Part 1: Fieldbus profiles*. International Electrotechnical Commission, 2007.

[ISO03a] ISO. *ISO 11898-1:2003: Road vehicles – Controller area network (CAN) – Part 1: Data link layer and physical signalling*. International Organization for Standardization, 2003.

[ISO03b] ISO. *ISO/IEC 14882:2003: Programming languages — C++*. International Organization for Standardization, 2003.

[JDHW09] Elmar Juergens, Florian Deissenboeck, Benjamin Hummel, and Stefan Wagner. Do code clones matter? In *Proceedings of the 31st International Conference on Software Engineering (ICSE'09)*, 2009.

[JGJ97] Ivar Jacobson, Martin L. Griss, and Patrik Jonsson. Making the reuse business work. *IEEE Computer*, 30(10):36–42, 1997.

[Kal98] Ferdinand Kallmeyer. *Eine Methode zur Modellierung prinzipieller Lösungen mechatronischer Systeme*. PhD thesis, Universität Paderborn, 1998.

[Kin95] Ekkart Kindler. *Collision detection in interactive 3D computer animation*. PhD thesis, Technische Universität München, 1995.

[KKWZ07] Roman Kontchakov, Agi Kurucz, Frank Wolter, and Michael Zakharyaschev. *Spatial Logic + Temporal Logic = ?*, pages 497–564. In Aiello et al. [APHvB07], 2007.

[KLM+97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP'97)*. Springer Verlag, 1997.

[Kos07] Rainer Koschke. Survey of research on software clones. In *Duplication, Redundancy, and Similarity in Software*, 2007.

[KWS+03] Oliver Kutz, Frank Wolter, Holger Sturm, Nobu-Yuki Suzuki, and Michael Zakharyaschev. Logics of metric spaces. *ACM Transactions on Computational Logic*, 4(2):260–294, 2003.

[Lip00] Christian Lippold. *Eine domänenübergreifende Konzeptionsumgebung für die Entwicklung mechatronischer Systeme*. PhD thesis, Ruhr-Universität Bochum, 2000.

[LSV03] Nancy A. Lynch, Roberto Segala, and Frits W. Vaandraager. Hybrid I/O automata. *Information and Computation*, 185(1):105–157, 2003.

[Lyn03] Nancy A. Lynch. Input/output automata: Basic, timed, hybrid, probabilistic, dynamic, ... In *Proceedings of the 14th International Conference on Concurrency Theory (CONCUR'03)*, 2003.

[McM93] Ken L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.

[Mil80] Robin Milner. *A Calculus of Communicating Systems*. Springer, 1980.

[Mos99] Pieter J. Mosterman. An overview of hybrid simulation phenomena and their support by simulation packages. In *Proceedings of the Second International Workshop on Hybrid Systems (HSCC'99)*, 1999.

[MS97] Olaf Müller and Peter Scholz. Functional specification of real-time and hybrid systems. In *Proceedings of the International Workshop on Hybrid and Real-Time Systems (HART'97)*, 1997.

[MWLF03] Sayan Mitra, Yong Wang, Nancy A. Lynch, and Eric Feron. Safety verification of model helicopter controller using hybrid input/output automata. In *Proceedings of the 6th International Workshop of Hybrid Systems: Computation and Control (HSCC'03)*, 2003.

[NPW02] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*. Springer, 2002.

[Obj08] Object Management Group. OMG SysML specification v1.1, 2008.

[PA10] Shari Lawrence Pfleeger and Joanne M. Atlee. *Software Engineering*. Pearson, fourth edition, 2010.

[Pav01] Luis Pavez, editor. *STEP Datenmodelle zur Simulation mechatronischer Systeme*. 2001.

[Pay60] Henry Paynter. *Analysis and Design of Engineering Systems*. McGraw-Hill, 1960.

[Pet62] Carl Adam Petri. *Kommunikation mit Automaten*. PhD thesis, Universität Bonn, 1962.

[Pie02] Benjamin C. Pierce. *Types and programming languages*. MIT Press, 2002.

[PLEB07] Carlos Pacheco, Shuvendu K. Lahiri, Michael D. Ernst, and Thomas Ball. Feedback-directed random test generation. In *Proceedings of the 29th International Conference on Software Engineering (ICSE'07)*, 2007.

[RC07] Chanchal Kumar Roy and James R. Cordy. A survey on software clone detection research. Technical Report 2007-541, Queen's University, Kingston, Canada, 2007.

[Sel98] Bran Selic. Using UML for modeling complex real-time systems. In *Proceedings of the Workshop on Languages, Compilers, and Tools for Embedded Systems (LCTES'98)*, 1998.

[SGW94] Bran Selic, Garth Gullekson, and Paul T. Ward. *Real-Time Object-Oriented Modeling*. Wiley, 1994.

[Sie03] Siemens AG. SIMIT Komponententypeditor CTE – Bedienungshandbuch V5.0, 2003.

[SKSV08] Peter Struss, Axel Kather, Dominik Schneider, and Tobias Voigt. A compositional mathematical model of machines transporting rigid objects. In *Proceedings of the 18th European Conference on Artificial Intelligence (ECAI'08)*, 2008.

[Som06] Ian Sommerville. *Software Engineering*. Addison-Wesley, 8th edition, 2006.

[SPHP02] Bernhard Schätz, Alexander Pretschner, Franz Huber, and Jan Philipps. Model-based development of embedded systems. In *Advances in Object-Oriented Information Systems (OOIS 2002 Workshops)*, 2002.

[SR98] Bran Selic and James Rumbaugh. UML for modeling complex real-time systems. Technical report, ObjecTime, 1998. Now available via IBM at `http://www.ibm.com/developerworks/rational/library/139.html`.

[SRE+08] Thomas Strasser, Martijn Rooker, Gerhard Ebenhofer, Ingo Hegny, Monika Wenger, Christoph Sünder, Allan Martel, and Antonia Valentini. Multi-domain model-driven design of industrial automation and control systems. In *Proceedings of 13th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA'08)*, 2008.

[Sta01] Thomas M. Stauner. *Systematic Development of Hybrid Systems*. PhD thesis, Technische Universität München, 2001.

[SW07] Wilhelm Schäfer and Heike Wehrheim. The challenges of building advanced mechatronic systems. In *Proceedings of the Workshop on the Future of Software Engineering (FOSE'07)*, 2007.

[TH09] Judith Thyssen and Benjamin Hummel. Behavioral specification of reactive systems using stream-based I/O tables. In *Proceedings of the 7th IEEE International Conference on Software Engineering and Formal Methods (SEFM'09)*, 2009.

[Til01] Michael Tiller. *Introduction to Physical Modeling with Modelica*. Springer, 2001.

[Tol54] John Ronald Reuel Tolkien. *The Lord of the Rings*. Allen & Unwin, 1954.

[vdB94] Michael von der Beeck. A comparison of statecharts variants. In *Proceedings of the Third International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems (FTRTFT'94)*, 1994.

[vdB97] Gino van den Bergen. Efficient collision detection of complex deformable models using AABB trees. *Journal of Graphics, GPU, and Game Tools*, 2(4):1–14, 1997.

[vdB99] Gino van den Bergen. *Collision detection in interactive 3D computer animation*. PhD thesis, Eindhoven University of Technology, 1999.

[WZ05] Frank Wolter and Michael Zakharyaschev. A logic for metric and topology. *Journal of Symbolic Logic*, 70:795–828, 2005.

[ZWHL06] Michael F. Zäh, Georg Wünsch, Thomas Hensel, and Alexander Lindworsky. Nutzen der virtuellen Inbetriebnahme: Ein Experiment. *ZWF Zeitschrift für wirtschaftlichen Fabrikbetrieb*, 10:595–599, 2006.

[ZZP07] Sajeh Zairi, Belhassen Zouari, and Laurent Pitrac. A formal approach for the specification, verification and control of flexible manufacturing systems. In *Proceedings of the 12th IEEE Conference on Emerging Technologies and Factory Automation (ETFA'07)*, 2007.